```python
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# ****************************************************************************
# * Software: FPDF for python                                                *
# * Version:  1.7.1                                                          *
# * Date:     2010-09-10                                                     *
# * Last update: 2012-08-16                                                  *
# * License:  LGPL v3.0                                                      *
# *                                                                          *
# * Original Author (PHP):  Olivier PLATHEY 2004-12-31                       *
# * Ported to Python 2.4 by Max (maxpat78@yahoo.it) on 2006-05               *
# * Maintainer:  Mariano Reingart (reingart@gmail.com) et al since 2008 est. *
# * NOTE: 'I' and 'D' destinations are disabled, and simply print to STDOUT  *
# ****************************************************************************

from __future__ import division, with_statement

from datetime import datetime
from functools import wraps
import math
import errno
import os, sys, zlib, struct, re, tempfile, struct

from .ttfonts import TTFontFile
from .fonts import fpdf_charwidths
from .php import substr, sprintf, print_r, UTF8ToUTF16BE, UTF8StringToArray
from .py3k import PY3K, pickle, urlopen, BytesIO, Image, basestring, unicode, exception, b

# Global variables
FPDF_VERSION = '1.7.2'
FPDF_FONT_DIR = os.path.join(os.path.dirname(__file__),'font')
FPDF_CACHE_MODE = 0 # 0 - in same folder, 1 - none, 2 - hash
FPDF_CACHE_DIR = None
SYSTEM_TTFONTS = None

PAGE_FORMATS = {
    "a3": (841.89, 1190.55),
    "a4": (595.28, 841.89),
    "a5": (420.94, 595.28),
    "letter": (612, 792),
    "legal": (612, 1008),
}

```

```
def set_global(var, val):
    globals()[var] = val

def load_cache(filename):
    """Return unpickled object, or None if cache unavailable"""
    if not filename:
        return None
    try:
        with open(filename, "rb") as fh:
            return pickle.load(fh)
    except (IOError, ValueError):  # File missing, unsupported pickle, etc
        return None

class FPDF(object):
    def __init__(self, orientation = 'P', unit = 'mm', format = 'A4'):
        # Some checks
        self._dochecks()
        # Initialization of properties
        self.offsets = {}                 # array of object offsets
        self.page = 0                     # current page number
        self.n = 2                        # current object number
        self.buffer = ''                  # buffer holding in-memory PDF
        self.pages = {}                   # array containing pages and metadata
        self.state = 0                    # current document state
        self.fonts = {}                   # array of used fonts
        self.font_files = {}              # array of font files
        self.diffs = {}                   # array of encoding differences
        self.images = {}                  # array of used images
        self.page_links = {}              # array of links in pages
        self.links = {}                   # array of internal links
        self.in_footer = 0                # flag set when processing footer
        self.lastw = 0
        self.lasth = 0                    # height of last cell printed
        self.font_family = ''             # current font family
        self.font_style = ''              # current font style
        self.font_size_pt = 12            # current font size in points
        self.font_stretching = 100        # current font stretching
        self.underline = 0                # underlining flag
        self.draw_color = '0 G'
        self.fill_color = '0 g'
        self.text_color = '0 g'
        self.color_flag = 0               # indicates whether fill and text colors are diffe
        self.ws = 0                       # word spacing
        self.angle = 0
        # Standard fonts
        self.core_fonts={'courier': 'Courier', 'courierB': 'Courier-Bold',
```

```
            'courierI': 'Courier-Oblique', 'courierBI': 'Courier-BoldOblique',
            'helvetica': 'Helvetica', 'helveticaB': 'Helvetica-Bold',
            'helveticaI': 'Helvetica-Oblique',
            'helveticaBI': 'Helvetica-BoldOblique',
            'times': 'Times-Roman', 'timesB': 'Times-Bold',
            'timesI': 'Times-Italic', 'timesBI': 'Times-BoldItalic',
            'symbol': 'Symbol', 'zapfdingbats': 'ZapfDingbats'}
        self.core_fonts_encoding = "latin-1"
        # Scale factor
        if unit == "pt":
            self.k = 1
        elif unit == "mm":
            self.k = 72 / 25.4
        elif unit == "cm":
            self.k = 72 / 2.54
        elif unit == 'in':
            self.k = 72.
        else:
            self.error("Incorrect unit: " + unit)
        # Page format
        self.fw_pt, self.fh_pt = self.get_page_format(format, self.k)
        self.dw_pt = self.fw_pt
        self.dh_pt = self.fh_pt
        self.fw = self.fw_pt / self.k
        self.fh = self.fh_pt / self.k
        # Page orientation
        orientation = orientation.lower()
        if orientation in ('p', 'portrait'):
            self.def_orientation = 'P'
            self.w_pt = self.fw_pt
            self.h_pt = self.fh_pt
        elif orientation in ('l', 'landscape'):
            self.def_orientation = 'L'
            self.w_pt = self.fh_pt
            self.h_pt = self.fw_pt
        else:
            self.error('Incorrect orientation: ' + orientation)
        self.cur_orientation = self.def_orientation
        self.w = self.w_pt / self.k
        self.h = self.h_pt / self.k
        # Page margins (1 cm)
        margin = 28.35 / self.k
        self.set_margins(margin, margin)
        # Interior cell margin (1 mm)
        self.c_margin = margin / 10.0
```

```
        # line width (0.2 mm)
        self.line_width = .567 / self.k
        # Automatic page break
        self.set_auto_page_break(1, 2 * margin)
        # Full width display mode
        self.set_display_mode('fullwidth')
        # Enable compression
        self.set_compression(1)
        # Set default PDF version number
        self.pdf_version = '1.3'
    def get_page_format(format, k):
        "Return scale factor, page w and h size in points"
        if isinstance(format, basestring):
            format = format.lower()
            if format in PAGE_FORMATS:
                return PAGE_FORMATS[format]
            else:
                raise RuntimeError("Unknown page format: " + format)
        else:
            return (format[0] * k, format[1] * k)
    def check_page(fn):
        "Decorator to protect drawing methods"
        @wraps(fn)
        def wrapper(self, *args, **kwargs):
            if not self.page and not kwargs.get('split_only'):
                self.error("No page open, you need to call add_page() first")
            else:
                return fn(self, *args, **kwargs)
        return wrapper
    def set_margins(self, left,top,right=-1):
        "Set left, top and right margins"
        self.l_margin=left
        self.t_margin=top
        if(right==-1):
            right=left
        self.r_margin=right
    def set_left_margin(self, margin):
        "Set left margin"
        self.l_margin=margin
        if(self.page>0 and self.x<margin):
            self.x=margin
```

```
184     def set_top_margin(self, margin):
185         "Set top margin"
186         self.t_margin=margin
188     def set_right_margin(self, margin):
189         "Set right margin"
190         self.r_margin=margin
192     def set_auto_page_break(self, auto,margin=0):
193         "Set auto page break mode and triggering margin"
194         self.auto_page_break=auto
195         self.b_margin=margin
196         self.page_break_trigger=self.h-margin
198     def set_display_mode(self, zoom,layout='continuous'):
199         """Set display mode in viewer
200 
201         The "zoom" argument may be 'fullpage', 'fullwidth', 'real',
202         'default', or a number, interpreted as a percentage."""
203 
204         if(zoom=='fullpage' or zoom=='fullwidth' or zoom=='real' or zoom=='default' or not
205             self.zoom_mode=zoom
206         else:
207             self.error('Incorrect zoom display mode: '+zoom)
208         if(layout=='single' or layout=='continuous' or layout=='two' or layout=='default')
209             self.layout_mode=layout
210         else:
211             self.error('Incorrect layout display mode: '+layout)
213     def set_compression(self, compress):
214         "Set page compression"
215         self.compress=compress
217     def set_title(self, title):
218         "Title of document"
219         self.title=title
221     def set_subject(self, subject):
222         "Subject of document"
223         self.subject=subject
225     def set_author(self, author):
226         "Author of document"
227         self.author=author
229     def set_keywords(self, keywords):
230         "Keywords of document"
231         self.keywords=keywords
```

```
    def set_creator(self, creator):
        "Creator of document"
        self.creator=creator
    def set_doc_option(self, opt, value):
        "Set document option"
        if opt == "core_fonts_encoding":
            self.core_fonts_encoding = value
        else:
            self.error("Unknown document option \"%s\"" % str(opt))
    def alias_nb_pages(self, alias='{nb}'):
        "Define an alias for total number of pages"
        self.str_alias_nb_pages=alias
        return alias
    def error(self, msg):
        "Fatal error"
        raise RuntimeError('FPDF error: '+msg)
    def open(self):
        "Begin document"
        self.state=1
    def close(self):
        "Terminate document"
        if(self.state==3):
            return
        if(self.page==0):
            self.add_page()
        #Page footer
        self.in_footer=1
        self.footer()
        self.in_footer=0
        #close page
        self._endpage()
        #close document
        self._enddoc()
    def add_page(self, orientation = '', format = '', same = False):
        "Start a new page, if same page format will be same as previous"
        if(self.state==0):
            self.open()
        family=self.font_family
        if self.underline:
            style = self.font_style + 'U'
        else:
            style = self.font_style
        size=self.font_size_pt
```

```
        lw=self.line_width
        dc=self.draw_color
        fc=self.fill_color
        tc=self.text_color
        cf=self.color_flag
        stretching=self.font_stretching
        if(self.page>0):
            #Page footer
            self.in_footer=1
            self.footer()
            self.in_footer=0
            #close page
            self._endpage()
        #Start new page
        self._beginpage(orientation, format, same)
        #Set line cap style to square
        self._out('2 J')
        #Set line width
        self.line_width=lw
        self._out(sprintf('%.2f w',lw*self.k))
        #Set font
        if(family):
            self.set_font(family,style,size)
        #Set colors
        self.draw_color=dc
        if(dc!='0 G'):
            self._out(dc)
        self.fill_color=fc
        if(fc!='0 g'):
            self._out(fc)
        self.text_color=tc
        self.color_flag=cf
        #Page header
        self.header()
        #Restore line width
        if(self.line_width!=lw):
            self.line_width=lw
            self._out(sprintf('%.2f w',lw*self.k))
        #Restore font
        if(family):
            self.set_font(family,style,size)
        #Restore colors
        if(self.draw_color!=dc):
            self.draw_color=dc
            self._out(dc)
```

```
327         if(self.fill_color!=fc):
328             self.fill_color=fc
329             self._out(fc)
330         self.text_color=tc
331         self.color_flag=cf
332         #Restore stretching
333         if(stretching != 100):
334             self.set_stretching(stretching)
336     def header(self):
337         "Header to be implemented in your own inherited class"
338         pass
340     def footer(self):
341         "Footer to be implemented in your own inherited class"
342         pass
344     def page_no(self):
345         "Get current page number"
346         return self.page
348     def set_draw_color(self, r,g=-1,b=-1):
349         "Set color for all stroking operations"
350         if((r==0 and g==0 and b==0) or g==-1):
351             self.draw_color=sprintf('%.3f G',r/255.0)
352         else:
353             self.draw_color=sprintf('%.3f %.3f %.3f RG',r/255.0,g/255.0,b/255.0)
354         if(self.page>0):
355             self._out(self.draw_color)
357     def set_fill_color(self,r,g=-1,b=-1):
358         "Set color for all filling operations"
359         if((r==0 and g==0 and b==0) or g==-1):
360             self.fill_color=sprintf('%.3f g',r/255.0)
361         else:
362             self.fill_color=sprintf('%.3f %.3f %.3f rg',r/255.0,g/255.0,b/255.0)
363         self.color_flag=(self.fill_color!=self.text_color)
364         if(self.page>0):
365             self._out(self.fill_color)
367     def set_text_color(self, r,g=-1,b=-1):
368         "Set color for text"
369         if((r==0 and g==0 and b==0) or g==-1):
370             self.text_color=sprintf('%.3f g',r/255.0)
371         else:
372             self.text_color=sprintf('%.3f %.3f %.3f rg',r/255.0,g/255.0,b/255.0)
373         self.color_flag=(self.fill_color!=self.text_color)
```

```
375     def get_string_width(self, s, normalized = False):
376         "Get width of a string in the current font"
377         # normalized is parameter for internal use
378         s = s if normalized else self.normalize_text(s)
379         cw=self.current_font['cw']
380         w=0
381         l=len(s)
382         if self.unifontsubset:
383             for char in s:
384                 char = ord(char)
385                 if len(cw) > char:
386                     w += cw[char] # ord(cw[2*char])<<8 + ord(cw[2*char+1])
387                 #elif (char>0 and char<128 and isset($cw[chr($char)])) { $w += $cw[chr($ch
388                 elif (self.current_font['desc']['MissingWidth']) :
389                     w += self.current_font['desc']['MissingWidth']
390                 #elif (isset($this->CurrentFont['MissingWidth'])) { $w += $this->CurrentFo
391                 else:
392                     w += 500
393         else:
394             for i in range(0, l):
395                 w += cw.get(s[i],0)
396         if self.font_stretching != 100:
397             w = w * self.font_stretching / 100.0
398         return w * self.font_size / 1000.0
400     def set_line_width(self, width):
401         "Set line width"
402         self.line_width=width
403         if(self.page>0):
404             self._out(sprintf('%.2f w',width*self.k))
407     def line(self, x1,y1,x2,y2):
408         "Draw a line"
409         self._out(sprintf('%.2f %.2f m %.2f %.2f l S',x1*self.k,(self.h-y1)*self.k,x2*self
411     def _set_dash(self, dash_length=False, space_length=False):
412         if(dash_length and space_length):
413             s = sprintf('[%.3f %.3f] 0 d', dash_length*self.k, space_length*self.k)
414         else:
415             s = '[] 0 d'
416         self._out(s)
```

```
419     def dashed_line(self, x1,y1,x2,y2, dash_length=1, space_length=1):
420         """Draw a dashed line. Same interface as line() except:
421            - dash_length: Length of the dash
422            - space_length: Length of the space between dashes"""
423         self._set_dash(dash_length, space_length)
424         self.line(x1, y1, x2, y2)
425         self._set_dash()
428     def rect(self, x,y,w,h,style=''):
429         "Draw a rectangle"
430         if(style=='F'):
431             op='f'
432         elif(style=='FD' or style=='DF'):
433             op='B'
434         else:
435             op='S'
436         self._out(sprintf('%.2f %.2f %.2f %.2f re %s',x*self.k,(self.h-y)*self.k,w*self.k,
```

```
    def ellipse(self, x,y,w,h,style=''):
        "Draw a ellipse"
        if(style=='F'):
            op='f'
        elif(style=='FD' or style=='DF'):
            op='B'
        else:
            op='S'

        cx = x + w/2.0
        cy = y + h/2.0
        rx = w/2.0
        ry = h/2.0

        lx = 4.0/3.0*(math.sqrt(2)-1)*rx
        ly = 4.0/3.0*(math.sqrt(2)-1)*ry

        self._out(sprintf('%.2f %.2f m %.2f %.2f %.2f %.2f %.2f %.2f c',
            (cx+rx)*self.k, (self.h-cy)*self.k,
            (cx+rx)*self.k, (self.h-(cy-ly))*self.k,
            (cx+lx)*self.k, (self.h-(cy-ry))*self.k,
            cx*self.k, (self.h-(cy-ry))*self.k))
        self._out(sprintf('%.2f %.2f %.2f %.2f %.2f %.2f c',
            (cx-lx)*self.k, (self.h-(cy-ry))*self.k,
            (cx-rx)*self.k, (self.h-(cy-ly))*self.k,
            (cx-rx)*self.k, (self.h-cy)*self.k))
        self._out(sprintf('%.2f %.2f %.2f %.2f %.2f %.2f c',
            (cx-rx)*self.k, (self.h-(cy+ly))*self.k,
            (cx-lx)*self.k, (self.h-(cy+ry))*self.k,
            cx*self.k, (self.h-(cy+ry))*self.k))
        self._out(sprintf('%.2f %.2f %.2f %.2f %.2f %.2f c %s',
            (cx+lx)*self.k, (self.h-(cy+ry))*self.k,
            (cx+rx)*self.k, (self.h-(cy+ly))*self.k,
            (cx+rx)*self.k, (self.h-cy)*self.k,
            op))
    def add_font(self, family, style='', fname='', uni=False):
        "Add a TrueType or Type1 font"
        family = family.lower()
        if (fname == ''):
            fname = family.replace(' ','') + style.lower() + '.pkl'
        if (family == 'arial'):
            family = 'helvetica'
        style = style.upper()
        if (style == 'IB'):
```

```
            style = 'BI'
        fontkey = family+style
        if fontkey in self.fonts:
            # Font already added!
            return
        if (uni):
            global SYSTEM_TTFONTS, FPDF_CACHE_MODE, FPDF_CACHE_DIR
            if os.path.exists(fname):
                ttffilename = fname
            elif (FPDF_FONT_DIR and
                os.path.exists(os.path.join(FPDF_FONT_DIR, fname))):
                ttffilename = os.path.join(FPDF_FONT_DIR, fname)
            elif (SYSTEM_TTFONTS and
                os.path.exists(os.path.join(SYSTEM_TTFONTS, fname))):
                ttffilename = os.path.join(SYSTEM_TTFONTS, fname)
            else:
                raise RuntimeError("TTF Font file not found: %s" % fname)
            name = ''
            if FPDF_CACHE_MODE == 0:
                unifilename = os.path.splitext(ttffilename)[0] + '.pkl'
            elif FPDF_CACHE_MODE == 2:
                unifilename = os.path.join(FPDF_CACHE_DIR, \
                    hashpath(ttffilename) + ".pkl")
            else:
                unifilename = None
            font_dict = load_cache(unifilename)
            if font_dict is None:
                ttf = TTFontFile()
                ttf.getMetrics(ttffilename)
                desc = {
                    'Ascent': int(round(ttf.ascent, 0)),
                    'Descent': int(round(ttf.descent, 0)),
                    'CapHeight': int(round(ttf.capHeight, 0)),
                    'Flags': ttf.flags,
                    'FontBBox': "[%s %s %s %s]" % (
                        int(round(ttf.bbox[0], 0)),
                        int(round(ttf.bbox[1], 0)),
                        int(round(ttf.bbox[2], 0)),
                        int(round(ttf.bbox[3], 0))),
                    'ItalicAngle': int(ttf.italicAngle),
                    'StemV': int(round(ttf.stemV, 0)),
                    'MissingWidth': int(round(ttf.defaultWidth, 0)),
                    }
                # Generate metrics .pkl file
                font_dict = {
```

```
                    'name': re.sub('[ ()]', '', ttf.fullName),
                    'type': 'TTF',
                    'desc': desc,
                    'up': round(ttf.underlinePosition),
                    'ut': round(ttf.underlineThickness),
                    'ttffile': ttffilename,
                    'fontkey': fontkey,
                    'originalsize': os.stat(ttffilename).st_size,
                    'cw': ttf.charWidths,
                    }
                if unifilename:
                    try:
                        with open(unifilename, "wb") as fh:
                            pickle.dump(font_dict, fh)
                    except IOError:
                        if not exception().errno == errno.EACCES:
                            raise  # Not a permission error.
                del ttf
            if hasattr(self,'str_alias_nb_pages'):
                sbarr = list(range(0,57))   # include numbers in the subset!
            else:
                sbarr = list(range(0,32))
            self.fonts[fontkey] = {
                'i': len(self.fonts)+1, 'type': font_dict['type'],
                'name': font_dict['name'], 'desc': font_dict['desc'],
                'up': font_dict['up'], 'ut': font_dict['ut'],
                'cw': font_dict['cw'],
                'ttffile': font_dict['ttffile'], 'fontkey': fontkey,
                'subset': sbarr, 'unifilename': unifilename,
                }
            self.font_files[fontkey] = {'length1': font_dict['originalsize'],
                                        'type': "TTF", 'ttffile': ttffilename}
            self.font_files[fname] = {'type': "TTF"}
        else:
            with open(fname, 'rb') as fontfile:
                font_dict = pickle.load(fontfile)
            self.fonts[fontkey] = {'i': len(self.fonts)+1}
            self.fonts[fontkey].update(font_dict)
            diff = font_dict.get('diff')
            if (diff):
                #Search existing encodings
                d = 0
                nb = len(self.diffs)
                for i in range(1, nb+1):
                    if(self.diffs[i] == diff):
```

```
                        d = i
                        break
                if (d == 0):
                    d = nb + 1
                    self.diffs[d] = diff
                self.fonts[fontkey]['diff'] = d
            filename = font_dict.get('filename')
            if (filename):
                if (font_dict['type'] == 'TrueType'):
                    originalsize = font_dict['originalsize']
                    self.font_files[filename]={'length1': originalsize}
                else:
                    self.font_files[filename]={'length1': font_dict['size1'],
                                               'length2': font_dict['size2']}
    def set_font(self, family,style='',size=0):
        "Select a font; size given in points"
        family=family.lower()
        if(family==''):
            family=self.font_family
        if(family=='arial'):
            family='helvetica'
        elif(family=='symbol' or family=='zapfdingbats'):
            style=''
        style=style.upper()
        if('U' in style):
            self.underline=1
            style=style.replace('U','')
        else:
            self.underline=0
        if(style=='IB'):
            style='BI'
        if(size==0):
            size=self.font_size_pt
        #Test if font is already selected
        if(self.font_family==family and self.font_style==style and self.font_size_pt==size
            return
        #Test if used for the first time
        fontkey=family+style
        if fontkey not in self.fonts:
            #Check if one of the standard fonts
            if fontkey in self.core_fonts:
                if fontkey not in fpdf_charwidths:
                    #Load metric file
                    name=os.path.join(FPDF_FONT_DIR,family)
```

```
                    if(family=='times' or family=='helvetica'):
                        name+=style.lower()
                    with open(name+'.font') as file:
                        exec(compile(file.read(), name+'.font', 'exec'))
                    if fontkey not in fpdf_charwidths:
                        self.error('Could not include font metric file for'+fontkey)
                i=len(self.fonts)+1
                self.fonts[fontkey]={'i':i,'type':'core','name':self.core_fonts[fontkey],'
            else:
                self.error('Undefined font: '+family+' '+style)
        #Select it
        self.font_family=family
        self.font_style=style
        self.font_size_pt=size
        self.font_size=size/self.k
        self.current_font=self.fonts[fontkey]
        self.unifontsubset = (self.fonts[fontkey]['type'] == 'TTF')
        if(self.page>0):
            self._out(sprintf('BT /F%d %.2f Tf ET',self.current_font['i'],self.font_size_p
    def set_font_size(self, size):
        "Set font size in points"
        if(self.font_size_pt==size):
            return
        self.font_size_pt=size
        self.font_size=size/self.k
        if(self.page>0):
            self._out(sprintf('BT /F%d %.2f Tf ET',self.current_font['i'],self.font_size_p
    def set_stretching(self, factor):
        "Set from stretch factor percents (default: 100.0)"
        if(self.font_stretching == factor):
            return
        self.font_stretching = factor
        if (self.page > 0):
            self._out(sprintf('BT %.2f Tz ET', self.font_stretching))
    def add_link(self):
        "Create a new internal link"
        n=len(self.links)+1
        self.links[n]=(0,0)
        return n
```

```
    def set_link(self, link,y=0,page=-1):
        "Set destination of internal link"
        if(y==-1):
            y=self.y
        if(page==-1):
            page=self.page
        self.links[link]=[page,y]

    def link(self, x,y,w,h,link):
        "Put a link on the page"
        if not self.page in self.page_links:
            self.page_links[self.page] = []
        self.page_links[self.page] += [(x*self.k,self.h_pt-y*self.k,w*self.k,h*self.k,link

    def text(self, x, y, txt=''):
        "Output a string"
        txt = self.normalize_text(txt)
        if (self.unifontsubset):
            txt2 = self._escape(UTF8ToUTF16BE(txt, False))
            for uni in UTF8StringToArray(txt):
                self.current_font['subset'].append(uni)
        else:
            txt2 = self._escape(txt)
        s=sprintf('BT %.2f %.2f Td (%s) Tj ET',x*self.k,(self.h-y)*self.k, txt2)
        if(self.underline and txt!=''):
            s+=' '+self._dounderline(x,y,txt)
        if(self.color_flag):
            s='q '+self.text_color+' '+s+' Q'
        self._out(s)

    def rotate(self, angle, x=None, y=None):
        if x is None:
            x = self.x
        if y is None:
            y = self.y;
        if self.angle!=0:
            self._out('Q')
        self.angle = angle
        if angle!=0:
            angle *= math.pi/180;
            c = math.cos(angle);
            s = math.sin(angle);
            cx = x*self.k;
            cy = (self.h-y)*self.k
            s = sprintf('q %.5F %.5F %.5F %.5F %.2F %.2F cm 1 0 0 1 %.2F %.2F cm',c,s,-s,c
            self._out(s)
```

```
711     def accept_page_break(self):
712         "Accept automatic page break or not"
713         return self.auto_page_break
716     def cell(self, w,h=0,txt='',border=0,ln=0,align='',fill=0,link=''):
717         "Output a cell"
718         txt = self.normalize_text(txt)
719         k=self.k
720         if(self.y+h>self.page_break_trigger and not self.in_footer and self.accept_page_br
721             #Automatic page break
722             x=self.x
723             ws=self.ws
724             if(ws>0):
725                 self.ws=0
726                 self._out('0 Tw')
727             self.add_page(same = True)
728             self.x=x
729             if(ws>0):
730                 self.ws=ws
731                 self._out(sprintf('%.3f Tw',ws*k))
732         if(w==0):
733             w=self.w-self.r_margin-self.x
734         s=''
735         if(fill==1 or border==1):
736             if(fill==1):
737                 if border==1:
738                     op='B'
739                 else:
740                     op='f'
741             else:
742                 op='S'
743             s=sprintf('%.2f %.2f %.2f %.2f re %s ',self.x*k,(self.h-self.y)*k,w*k,-h*k,op)
744         if(isinstance(border,basestring)):
745             x=self.x
746             y=self.y
747             if('L' in border):
748                 s+=sprintf('%.2f %.2f m %.2f %.2f l S ',x*k,(self.h-y)*k,x*k,(self.h-(y+h)
749             if('T' in border):
750                 s+=sprintf('%.2f %.2f m %.2f %.2f l S ',x*k,(self.h-y)*k,(x+w)*k,(self.h-y
751             if('R' in border):
752                 s+=sprintf('%.2f %.2f m %.2f %.2f l S ',(x+w)*k,(self.h-y)*k,(x+w)*k,(self
753             if('B' in border):
754                 s+=sprintf('%.2f %.2f m %.2f %.2f l S ',x*k,(self.h-(y+h))*k,(x+w)*k,(self
755         if(txt!=''):
756             if(align=='R'):
```

```
            dx=w-self.c_margin-self.get_string_width(txt, True)
        elif(align=='C'):
            dx=(w-self.get_string_width(txt, True))/2.0
        else:
            dx=self.c_margin
        if(self.color_flag):
            s+='q '+self.text_color+' '

        # If multibyte, Tw has no effect - do word spacing using an adjustment before
        if (self.ws and self.unifontsubset):
            for uni in UTF8StringToArray(txt):
                self.current_font['subset'].append(uni)
            space = self._escape(UTF8ToUTF16BE(' ', False))
            s += sprintf('BT 0 Tw %.2F %.2F Td [',(self.x + dx) * k,(self.h - (self.y
            t = txt.split(' ')
            numt = len(t)
            for i in range(numt):
                tx = t[i]
                tx = '(' + self._escape(UTF8ToUTF16BE(tx, False)) + ')'
                s += sprintf('%s ', tx);
                if ((i+1)<numt):
                    adj = -(self.ws * self.k) * 1000 / self.font_size_pt
                    s += sprintf('%d(%s) ', adj, space)
            s += '] TJ'
            s += ' ET'
        else:
            if (self.unifontsubset):
                txt2 = self._escape(UTF8ToUTF16BE(txt, False))
                for uni in UTF8StringToArray(txt):
                    self.current_font['subset'].append(uni)
            else:
                txt2 = self._escape(txt)
            s += sprintf('BT %.2f %.2f Td (%s) Tj ET',(self.x+dx)*k,(self.h-(self.y+.5

        if(self.underline):
            s+=' '+self._dounderline(self.x+dx,self.y+.5*h+.3*self.font_size,txt)
        if(self.color_flag):
            s+=' Q'
        if(link):
            self.link(self.x+dx,self.y+.5*h-.5*self.font_size,self.get_string_width(tx
    if(s):
        self._out(s)
    self.lasth=h
    if(ln>0):
        #Go to next line
```

```
            self.y+=h
            if(ln==1):
                self.x=self.l_margin
        else:
            self.x+=w
    def multi_cell(self, w, h, txt='', border=0, align='J', fill=0, split_only=False):
        "Output text with automatic or explicit line breaks"
        txt = self.normalize_text(txt)
        ret = [] # if split_only = True, returns splited text cells
        cw=self.current_font['cw']
        if(w==0):
            w=self.w-self.r_margin-self.x
        wmax=(w-2*self.c_margin)*1000.0/self.font_size
        s=txt.replace("\r",'')
        nb=len(s)
        if(nb>0 and s[nb-1]=="\n"):
            nb-=1
        b=0
        if(border):
            if(border==1):
                border='LTRB'
                b='LRT'
                b2='LR'
            else:
                b2=''
                if('L' in border):
                    b2+='L'
                if('R' in border):
                    b2+='R'
                if ('T' in border):
                    b=b2+'T'
                else:
                    b=b2
        sep=-1
        i=0
        j=0
        l=0
        ns=0
        nl=1
        while(i<nb):
            #Get next character
            c=s[i]
            if(c=="\n"):
                #Explicit line break
```

```
                if(self.ws>0):
                    self.ws=0
                    if not split_only:
                        self._out('0 Tw')
                if not split_only:
                    self.cell(w,h,substr(s,j,i-j),b,2,align,fill)
                else:
                    ret.append(substr(s,j,i-j))
                i+=1
                sep=-1
                j=i
                l=0
                ns=0
                nl+=1
                if(border and nl==2):
                    b=b2
                continue
            if(c==' '):
                sep=i
                ls=l
                ns+=1
            if self.unifontsubset:
                l += self.get_string_width(c, True) / self.font_size*1000.0
            else:
                l += cw.get(c,0)
            if(l>wmax):
                #Automatic line break
                if(sep==-1):
                    if(i==j):
                        i+=1
                    if(self.ws>0):
                        self.ws=0
                        if not split_only:
                            self._out('0 Tw')
                    if not split_only:
                        self.cell(w,h,substr(s,j,i-j),b,2,align,fill)
                    else:
                        ret.append(substr(s,j,i-j))
                else:
                    if(align=='J'):
                        if ns>1:
                            self.ws=(wmax-ls)/1000.0*self.font_size/(ns-1)
                        else:
                            self.ws=0
                        if not split_only:
```

```
                        self._out(sprintf('%.3f Tw',self.ws*self.k))
                    if not split_only:
                        self.cell(w,h,substr(s,j,sep-j),b,2,align,fill)
                    else:
                        ret.append(substr(s,j,sep-j))
                    i=sep+1
                sep=-1
                j=i
                l=0
                ns=0
                nl+=1
                if(border and nl==2):
                    b=b2
            else:
                i+=1
        #Last chunk
        if(self.ws>0):
            self.ws=0
            if not split_only:
                self._out('0 Tw')
        if(border and 'B' in border):
            b+='B'
        if not split_only:
            self.cell(w,h,substr(s,j,i-j),b,2,align,fill)
            self.x=self.l_margin
        else:
            ret.append(substr(s,j,i-j))
        return ret
    def write(self, h, txt='', link=''):
        "Output text in flowing mode"
        txt = self.normalize_text(txt)
        cw=self.current_font['cw']
        w=self.w-self.r_margin-self.x
        wmax=(w-2*self.c_margin)*1000.0/self.font_size
        s=txt.replace("\r",'')
        nb=len(s)
        sep=-1
        i=0
        j=0
        l=0
        nl=1
        while(i<nb):
            #Get next character
            c=s[i]
```

```
            if(c=="\n"):
                #Explicit line break
                self.cell(w,h,substr(s,j,i-j),0,2,'',0,link)
                i+=1
                sep=-1
                j=i
                l=0
                if(nl==1):
                    self.x=self.l_margin
                    w=self.w-self.r_margin-self.x
                    wmax=(w-2*self.c_margin)*1000.0/self.font_size
                nl+=1
                continue
            if(c==' '):
                sep=i
            if self.unifontsubset:
                l += self.get_string_width(c, True) / self.font_size*1000.0
            else:
                l += cw.get(c,0)
            if(l>wmax):
                #Automatic line break
                if(sep==-1):
                    if(self.x>self.l_margin):
                        #Move to next line
                        self.x=self.l_margin
                        self.y+=h
                        w=self.w-self.r_margin-self.x
                        wmax=(w-2*self.c_margin)*1000.0/self.font_size
                        i+=1
                        nl+=1
                        continue
                    if(i==j):
                        i+=1
                    self.cell(w,h,substr(s,j,i-j),0,2,'',0,link)
                else:
                    self.cell(w,h,substr(s,j,sep-j),0,2,'',0,link)
                    i=sep+1
                sep=-1
                j=i
                l=0
                if(nl==1):
                    self.x=self.l_margin
                    w=self.w-self.r_margin-self.x
                    wmax=(w-2*self.c_margin)*1000.0/self.font_size
                nl+=1
```

```
            else:
                i+=1
        #Last chunk
        if(i!=j):
            self.cell(l/1000.0*self.font_size,h,substr(s,j),0,0,'',0,link)
```

```
    def image(self, name, x=None, y=None, w=0,h=0,type='',link='', is_mask=False, mask_ima
        "Put an image on the page"
        if not name in self.images:
            #First use of image, get info
            if(type==''):
                pos=name.rfind('.')
                if(not pos):
                    self.error('image file has no extension and no type was specified: '+n
                type=substr(name,pos+1)
            type=type.lower()
            if(type=='jpg' or type=='jpeg'):
                info=self._parsejpg(name)
            elif(type=='png'):
                info=self._parsepng(name)
            else:
                #Allow for additional formats
                #maybe the image is not showing the correct extension,
                #but the header is OK,
                succeed_parsing = False
                #try all the parsing functions
                parsing_functions = [self._parsejpg,self._parsepng,self._parsegif]
                for pf in parsing_functions:
                    try:
                        info = pf(name)
                        succeed_parsing = True
                        break;
                    except:
                        pass
                #last resource
                if not succeed_parsing:
                    mtd='_parse'+type
                    if not hasattr(self,mtd):
                        self.error('Unsupported image type: '+type)
                    info=getattr(self, mtd)(name)
                mtd='_parse'+type
                if not hasattr(self,mtd):
                    self.error('Unsupported image type: '+type)
                info=getattr(self, mtd)(name)
            info['i']=len(self.images)+1
```

```
            # is_mask and mask_image
            if is_mask and info['cs'] != 'DeviceGray':
                self.error('Mask must be a gray scale image')
            if mask_image:
                info['masked'] = mask_image
            self.images[name]=info
        else:
            info=self.images[name]
        #Automatic width and height calculation if needed
        if(w==0 and h==0):
            #Put image at 72 dpi
            w=info['w']/self.k
            h=info['h']/self.k
        elif(w==0):
            w=h*info['w']/info['h']
        elif(h==0):
            h=w*info['h']/info['w']
        # Flowing mode
        if y is None:
            if (self.y + h > self.page_break_trigger and not self.in_footer and self.accep
                #Automatic page break
                x = self.x
                self.add_page(same = True)
                self.x = x
            y = self.y
            self.y += h
        if x is None:
            x = self.x
        if not is_mask:
            self._out(sprintf('q %.2f 0 0 %.2f %.2f %.2f cm /I%d Do Q',w*self.k,h*self.k,x
        if(link):
            self.link(x,y,w,h,link)

        return info
    def ln(self, h=''):
        "Line Feed; default value is last cell height"
        self.x=self.l_margin
        if(isinstance(h, basestring)):
            self.y+=self.lasth
        else:
            self.y+=h
    def get_x(self):
        "Get x position"
        return self.x
```

```
    def set_x(self, x):
        "Set x position"
        if(x>=0):
            self.x=x
        else:
            self.x=self.w+x
    def get_y(self):
        "Get y position"
        return self.y
    def set_y(self, y):
        "Set y position and reset x"
        self.x=self.l_margin
        if(y>=0):
            self.y=y
        else:
            self.y=self.h+y
    def set_xy(self, x,y):
        "Set x and y positions"
        self.set_y(y)
        self.set_x(x)
```

```
1102     def output(self, name='',dest=''):
1103         """Output PDF to some destination
1104 
1105         By default the PDF is written to sys.stdout. If a name is given, the
1106         PDF is written to a new file. If dest='S' is given, the PDF data is
1107         returned as a byte string."""
1108 
1109         #Finish document if necessary
1110         if(self.state<3):
1111             self.close()
1112         dest=dest.upper()
1113         if(dest==''):
1114             if(name==''):
1115                 dest='I'
1116             else:
1117                 dest='F'
1118         if PY3K:
1119             # manage binary data as latin1 until PEP461 or similar is implemented
1120             buffer = self.buffer.encode("latin1")
1121         else:
1122             buffer = self.buffer
1123         if dest in ('I', 'D'):
1124             # Python < 3 writes byte data transparently without "buffer"
1125             stdout = getattr(sys.stdout, 'buffer', sys.stdout)
1126             stdout.write(buffer)
1127         elif dest=='F':
1128             #Save to local file
1129             with open(name,'wb') as f:
1130                 f.write(buffer)
1131         elif dest=='S':
1132             #Return as a byte string
1133             return buffer
1134         else:
1135             self.error('Incorrect output destination: '+dest)
```

```
    def normalize_text(self, txt):
        "Check that text input is in the correct format/encoding"
        # - for TTF unicode fonts: unicode object (utf8 encoding)
        # - for built-in fonts: string instances (encoding: latin-1, cp1252)
        if not PY3K:
            if self.unifontsubset and isinstance(txt, str):
                return txt.decode("utf-8")
            elif not self.unifontsubset and isinstance(txt, unicode):
                return txt.encode(self.core_fonts_encoding)
        else:
            if not self.unifontsubset and self.core_fonts_encoding:
                return txt.encode(self.core_fonts_encoding).decode("latin-1")
        return txt
    def _dochecks(self):
        if(sprintf('%.1f',1.0)!='1.0'):
            import locale
            locale.setlocale(locale.LC_NUMERIC,'C')
    def _getfontpath(self):
        return FPDF_FONT_DIR+'/'
    def _putpages(self):
        nb = self.page
        if hasattr(self, 'str_alias_nb_pages'):
            # Replace number of pages in fonts using subsets (unicode)
            alias = UTF8ToUTF16BE(self.str_alias_nb_pages, False)
            r = UTF8ToUTF16BE(str(nb), False)
            for n in range(1, nb + 1):
                self.pages[n]["content"] = \
                    self.pages[n]["content"].replace(alias, r)
            # Now repeat for no pages in non-subset fonts
            for n in range(1,nb + 1):
                self.pages[n]["content"] = \
                    self.pages[n]["content"].replace(self.str_alias_nb_pages,
                        str(nb))
        if self.def_orientation == 'P':
            dw_pt = self.dw_pt
            dh_pt = self.dh_pt
        else:
            dw_pt = self.dh_pt
            dh_pt = self.dw_pt
        if self.compress:
            filter = '/Filter /FlateDecode '
        else:
            filter = ''
        for n in range(1, nb + 1):
```

```
            # Page
            self._newobj()
            self._out('<</Type /Page')
            self._out('/Parent 1 0 R')
            w_pt = self.pages[n]["w_pt"]
            h_pt = self.pages[n]["h_pt"]
            if w_pt != dw_pt or h_pt != dh_pt:
                self._out(sprintf('/MediaBox [0 0 %.2f %.2f]', w_pt, h_pt))
            self._out('/Resources 2 0 R')
            if self.page_links and n in self.page_links:
                # Links
                annots = '/Annots ['
                for pl in self.page_links[n]:
                    rect = sprintf('%.2f %.2f %.2f %.2f', pl[0], pl[1],
                        pl[0] + pl[2], pl[1] - pl[3])
                    annots += '<</Type /Annot /Subtype /Link /Rect [' + \
                        rect + '] /Border [0 0 0] '
                    if isinstance(pl[4], basestring):
                        annots += '/A <</S /URI /URI ' + \
                            self._textstring(pl[4]) + '>>>>'
                    else:
                        l = self.links[pl[4]]
                        if l[0] in self.orientation_changes:
                            h = w_pt
                        else:
                            h = h_pt
                        annots += sprintf('/Dest [%d 0 R /XYZ 0 %.2f null]>>',
                            1 + 2 * l[0], h - l[1] * self.k)
                self._out(annots + ']')
            if self.pdf_version > '1.3':
                self._out("/Group <</Type /Group /S /Transparency"\
                    "/CS /DeviceRGB>>")
            self._out('/Contents ' + str(self.n + 1) + ' 0 R>>')
            self._out('endobj')
            # Page content
            content = self.pages[n]["content"]
            if self.compress:
                # manage binary data as latin1 until PEP461 or similar is implemented
                p = content.encode("latin1") if PY3K else content
                p = zlib.compress(p)
            else:
                p = content
            self._newobj()
            self._out('<<' + filter + '/Length ' + str(len(p)) + '>>')
            self._putstream(p)
```

```
            self._out('endobj')
        # Pages root
        self.offsets[1] = len(self.buffer)
        self._out('1 0 obj')
        self._out('<</Type /Pages')
        kids = '/Kids ['
        for i in range(0, nb):
            kids += str(3 + 2 * i) + ' 0 R '
        self._out(kids + ']')
        self._out('/Count ' + str(nb))
        self._out(sprintf('/MediaBox [0 0 %.2f %.2f]', dw_pt, dh_pt))
        self._out('>>')
        self._out('endobj')
    def _putfonts(self):
        nf=self.n
        for diff in self.diffs:
            #Encodings
            self._newobj()
            self._out('<</Type /Encoding /BaseEncoding /WinAnsiEncoding /Differences ['+se
            self._out('endobj')
        for name,info in self.font_files.items():
            if 'type' in info and info['type'] != 'TTF':
                #Font file embedding
                self._newobj()
                self.font_files[name]['n']=self.n
                with open(self._getfontpath()+name,'rb',1) as f:
                    font=f.read()
                compressed=(substr(name,-2)=='.z')
                if(not compressed and 'length2' in info):
                    header=(ord(font[0])==128)
                    if(header):
                        #Strip first binary header
                        font=substr(font,6)
                    if(header and ord(font[info['length1']])==128):
                        #Strip second binary header
                        font=substr(font,0,info['length1'])+substr(font,info['length1']+6)
                self._out('<</Length '+str(len(font)))
                if(compressed):
                    self._out('/Filter /FlateDecode')
                self._out('/Length1 '+str(info['length1']))
                if('length2' in info):
                    self._out('/Length2 '+str(info['length2'])+' /Length3 0')
                self._out('>>')
                self._putstream(font)
```

```
                self._out('endobj')
        flist = [(x[1]["i"],x[0],x[1]) for x in self.fonts.items()]
        flist.sort()
        for idx,k,font in flist:
            #Font objects
            self.fonts[k]['n']=self.n+1
            type=font['type']
            name=font['name']
            if(type=='core'):
                #Standard font
                self._newobj()
                self._out('<</Type /Font')
                self._out('/BaseFont /'+name)
                self._out('/Subtype /Type1')
                if(name!='Symbol' and name!='ZapfDingbats'):
                    self._out('/Encoding /WinAnsiEncoding')
                self._out('>>')
                self._out('endobj')
            elif(type=='Type1' or type=='TrueType'):
                #Additional Type1 or TrueType font
                self._newobj()
                self._out('<</Type /Font')
                self._out('/BaseFont /'+name)
                self._out('/Subtype /'+type)
                self._out('/FirstChar 32 /LastChar 255')
                self._out('/Widths '+str(self.n+1)+' 0 R')
                self._out('/FontDescriptor '+str(self.n+2)+' 0 R')
                if(font['enc']):
                    if('diff' in font):
                        self._out('/Encoding '+str(nf+font['diff'])+' 0 R')
                    else:
                        self._out('/Encoding /WinAnsiEncoding')
                self._out('>>')
                self._out('endobj')
                #Widths
                self._newobj()
                cw=font['cw']
                s='['
                for i in range(32,256):
                    # Get doesn't raise exception; returns 0 instead of None if not set
                    s+=str(cw.get(chr(i)) or 0)+' '
                self._out(s+']')
                self._out('endobj')
                #Descriptor
                self._newobj()
```

```
1319                s='<</Type /FontDescriptor /FontName /'+name
1320                for k in ('Ascent', 'Descent', 'CapHeight', 'Flags', 'FontBBox', 'ItalicAn
1321                    s += ' /%s %s' % (k, font['desc'][k])
1322                filename=font['file']
1323                if(filename):
1324                    s+=' /FontFile'
1325                    if type!='Type1':
1326                        s+='2'
1327                    s+=' '+str(self.font_files[filename]['n'])+' 0 R'
1328                self._out(s+'>>')
1329                self._out('endobj')
1330            elif (type == 'TTF'):
1331                self.fonts[k]['n'] = self.n + 1
1332                ttf = TTFontFile()
1333                fontname = 'MPDFAA' + '+' + font['name']
1334                subset = font['subset']
1335                del subset[0]
1336                ttfontstream = ttf.makeSubset(font['ttffile'], subset)
1337                ttfontsize = len(ttfontstream)
1338                fontstream = zlib.compress(ttfontstream)
1339                codeToGlyph = ttf.codeToGlyph
1340                ##del codeToGlyph[0]
1341                # Type0 Font
1342                # A composite font - a font composed of other fonts, organized hierarchica
1343                self._newobj()
1344                self._out('<</Type /Font');
1345                self._out('/Subtype /Type0');
1346                self._out('/BaseFont /' + fontname + '');
1347                self._out('/Encoding /Identity-H');
1348                self._out('/DescendantFonts [' + str(self.n + 1) + ' 0 R]')
1349                self._out('/ToUnicode ' + str(self.n + 2) + ' 0 R')
1350                self._out('>>')
1351                self._out('endobj')
1352
1353                # CIDFontType2
1354                # A CIDFont whose glyph descriptions are based on TrueType font technology
1355                self._newobj()
1356                self._out('<</Type /Font')
1357                self._out('/Subtype /CIDFontType2')
1358                self._out('/BaseFont /' + fontname + '')
1359                self._out('/CIDSystemInfo ' + str(self.n + 2) + ' 0 R')
1360                self._out('/FontDescriptor ' + str(self.n + 3) + ' 0 R')
1361                if (font['desc'].get('MissingWidth')):
1362                    self._out('/DW %d' % font['desc']['MissingWidth'])
1363                self._putTTfontwidths(font, ttf.maxUni)
```

```
1364                self._out('/CIDToGIDMap ' + str(self.n + 4) + ' 0 R')
1365                self._out('>>')
1366                self._out('endobj')
1367
1368                # ToUnicode
1369                self._newobj()
1370                toUni = "/CIDInit /ProcSet findresource begin\n" \
1371                        "12 dict begin\n" \
1372                        "begincmap\n" \
1373                        "/CIDSystemInfo\n" \
1374                        "<</Registry (Adobe)\n" \
1375                        "/Ordering (UCS)\n" \
1376                        "/Supplement 0\n" \
1377                        ">> def\n" \
1378                        "/CMapName /Adobe-Identity-UCS def\n" \
1379                        "/CMapType 2 def\n" \
1380                        "1 begincodespacerange\n" \
1381                        "<0000> <FFFF>\n" \
1382                        "endcodespacerange\n" \
1383                        "1 beginbfrange\n" \
1384                        "<0000> <FFFF> <0000>\n" \
1385                        "endbfrange\n" \
1386                        "endcmap\n" \
1387                        "CMapName currentdict /CMap defineresource pop\n" \
1388                        "end\n" \
1389                        "end"
1390                self._out('<</Length ' + str(len(toUni)) + '>>')
1391                self._putstream(toUni)
1392                self._out('endobj')
1393
1394                # CIDSystemInfo dictionary
1395                self._newobj()
1396                self._out('<</Registry (Adobe)')
1397                self._out('/Ordering (UCS)')
1398                self._out('/Supplement 0')
1399                self._out('>>')
1400                self._out('endobj')
1401
1402                # Font descriptor
1403                self._newobj()
1404                self._out('<</Type /FontDescriptor')
1405                self._out('/FontName /' + fontname)
1406                for kd in ('Ascent', 'Descent', 'CapHeight', 'Flags', 'FontBBox', 'ItalicA
1407                    v = font['desc'][kd]
1408                    if (kd == 'Flags'):
```

```
                    v = v | 4;
                    v = v & ~32; # SYMBOLIC font flag
                self._out(' /%s %s' % (kd, v))
            self._out('/FontFile2 ' + str(self.n + 2) + ' 0 R')
            self._out('>>')
            self._out('endobj')

            # Embed CIDToGIDMap
            # A specification of the mapping from CIDs to glyph indices
            cidtogidmap = '';
            cidtogidmap = ["\x00"] * 256*256*2
            for cc, glyph in codeToGlyph.items():
                cidtogidmap[cc*2] = chr(glyph >> 8)
                cidtogidmap[cc*2 + 1] = chr(glyph & 0xFF)
            cidtogidmap = ''.join(cidtogidmap)
            if PY3K:
                # manage binary data as latin1 until PEP461-like function is implement
                cidtogidmap = cidtogidmap.encode("latin1")
            cidtogidmap = zlib.compress(cidtogidmap);
            self._newobj()
            self._out('<</Length ' + str(len(cidtogidmap)) + '')
            self._out('/Filter /FlateDecode')
            self._out('>>')
            self._putstream(cidtogidmap)
            self._out('endobj')

            #Font file
            self._newobj()
            self._out('<</Length ' + str(len(fontstream)))
            self._out('/Filter /FlateDecode')
            self._out('/Length1 ' + str(ttfontsize))
            self._out('>>')
            self._putstream(fontstream)
            self._out('endobj')
            del ttf
        else:
            #Allow for additional types
            mtd='_put'+type.lower()
            if(not method_exists(self,mtd)):
                self.error('Unsupported font type: '+type)
            self.mtd(font)

    def _putTTfontwidths(self, font, maxUni):
        if font['unifilename']:
            cw127fname = os.path.splitext(font['unifilename'])[0] + '.cw127.pkl'
```

```
        else:
            cw127fname = None
        font_dict = load_cache(cw127fname)
        if font_dict is None:
            rangeid = 0
            range_ = {}
            range_interval = {}
            prevcid = -2
            prevwidth = -1
            interval = False
            startcid = 1
        else:
            rangeid = font_dict['rangeid']
            range_ = font_dict['range']
            prevcid = font_dict['prevcid']
            prevwidth = font_dict['prevwidth']
            interval = font_dict['interval']
            range_interval = font_dict['range_interval']
            startcid = 128
        cwlen = maxUni + 1

        # for each character
        subset = set(font['subset'])
        for cid in range(startcid, cwlen):
            if cid == 128 and cw127fname and not os.path.exists(cw127fname):
                try:
                    with open(cw127fname, "wb") as fh:
                        font_dict = {}
                        font_dict['rangeid'] = rangeid
                        font_dict['prevcid'] = prevcid
                        font_dict['prevwidth'] = prevwidth
                        font_dict['interval'] = interval
                        font_dict['range_interval'] = range_interval
                        font_dict['range'] = range_
                        pickle.dump(font_dict, fh)
                except IOError:
                    if not exception().errno == errno.EACCES:
                        raise  # Not a permission error.
            if cid > 255 and (cid not in subset): #
                continue
            width = font['cw'][cid]
            if (width == 0):
                continue
            if (width == 65535): width = 0
            if ('dw' not in font or (font['dw'] and width != font['dw'])):
```

```
                if (cid == (prevcid + 1)):
                    if (width == prevwidth):
                        if (width == range_[rangeid][0]):
                            range_.setdefault(rangeid, []).append(width)
                        else:
                            range_[rangeid].pop()
                            # new range
                            rangeid = prevcid
                            range_[rangeid] = [prevwidth, width]
                        interval = True
                        range_interval[rangeid] = True
                    else:
                        if (interval):
                            # new range
                            rangeid = cid
                            range_[rangeid] = [width]
                        else:
                            range_[rangeid].append(width)
                        interval = False
                else:
                    rangeid = cid
                    range_[rangeid] = [width]
                    interval = False
                prevcid = cid
                prevwidth = width
        prevk = -1
        nextk = -1
        prevint = False
        for k, ws in sorted(range_.items()):
            cws = len(ws)
            if (k == nextk and not prevint and (not k in range_interval or cws < 3)):
                if (k in range_interval):
                    del range_interval[k]
                range_[prevk] = range_[prevk] + range_[k]
                del range_[k]
            else:
                prevk = k
            nextk = k + cws
            if (k in range_interval):
                prevint = (cws > 3)
                del range_interval[k]
                nextk -= 1
            else:
                prevint = False
        w = []
```

```
        for k, ws in sorted(range_.items()):
            if (len(set(ws)) == 1):
                w.append(' %s %s %s' % (k, k + len(ws) - 1, ws[0]))
            else:
                w.append(' %s [ %s ]\n' % (k, ' '.join([str(int(h)) for h in ws]))) ##
        self._out('/W [%s]' % ''.join(w))
    def _putimages(self):
        filter=''
        if self.compress:
            filter='/Filter /FlateDecode '
        i = [(x[1]["i"],x[1]) for x in self.images.items()]
        i.sort()
        for idx,info in i:
            self._putimage(info)
            del info['data']
            if 'smask' in info:
                del info['smask']
    def _putimage(self, info):
        if 'data' in info:
            self._newobj()
            info['n']=self.n
            self._out('<</Type /XObject')
            self._out('/Subtype /Image')
            self._out('/Width '+str(info['w']))
            self._out('/Height '+str(info['h']))
            # set mask object for this image
            if 'masked' in info:
                self._out('/SMask ' + str(info['masked']['n']+1) + ' 0 R')

            if(info['cs']=='Indexed'):
                self._out('/ColorSpace [/Indexed /DeviceRGB '+str(len(info['pal'])//3-1)+'
            else:
                self._out('/ColorSpace /'+info['cs'])
                if(info['cs']=='DeviceCMYK'):
                    self._out('/Decode [1 0 1 0 1 0 1 0]')
            self._out('/BitsPerComponent '+str(info['bpc']))
            if 'f' in info:
                self._out('/Filter /'+info['f'])
            if 'dp' in info:
                self._out('/DecodeParms <<' + info['dp'] + '>>')
            if('trns' in info and isinstance(info['trns'], list)):
                trns=''
                for i in range(0,len(info['trns'])):
                    trns+=str(info['trns'][i])+' '+str(info['trns'][i])+' '
```

```
                self._out('/Mask ['+trns+']')
            if('smask' in info):
                self._out('/SMask ' + str(self.n+1) + ' 0 R');
            self._out('/Length '+str(len(info['data']))+'>>')
            self._putstream(info['data'])
            self._out('endobj')
            # Soft mask
            if('smask' in info):
                dp = '/Predictor 15 /Colors 1 /BitsPerComponent 8 /Columns ' + str(info['w
                smask = {'w': info['w'], 'h': info['h'], 'cs': 'DeviceGray', 'bpc': 8, 'f'
                self._putimage(smask)
            #Palette
            if(info['cs']=='Indexed'):
                self._newobj()
                filter = self.compress and '/Filter /FlateDecode ' or ''
                if self.compress:
                    pal=zlib.compress(info['pal'])
                else:
                    pal=info['pal']
                self._out('<<'+filter+'/Length '+str(len(pal))+'>>')
                self._putstream(pal)
                self._out('endobj')
    def _putxobjectdict(self):
        i = [(x["i"],x["n"]) for x in self.images.values()]
        i.sort()
        for idx,n in i:
            self._out('/I'+str(idx)+' '+str(n)+' 0 R')
    def _putresourcedict(self):
        self._out('/ProcSet [/PDF /Text /ImageB /ImageC /ImageI]')
        self._out('/Font <<')
        f = [(x["i"],x["n"]) for x in self.fonts.values()]
        f.sort()
        for idx,n in f:
            self._out('/F'+str(idx)+' '+str(n)+' 0 R')
        self._out('>>')
        self._out('/XObject <<')
        self._putxobjectdict()
        self._out('>>')
```

```
1631    def _putresources(self):
1632        self._putfonts()
1633        self._putimages()
1634        #Resource dictionary
1635        self.offsets[2]=len(self.buffer)
1636        self._out('2 0 obj')
1637        self._out('<<')
1638        self._putresourcedict()
1639        self._out('>>')
1640        self._out('endobj')
1642    def _putinfo(self):
1643        self._out('/Producer '+self._textstring('PyFPDF '+FPDF_VERSION+' http://pyfpdf.goc
1644        if hasattr(self,'title'):
1645            self._out('/Title '+self._textstring(self.title))
1646        if hasattr(self,'subject'):
1647            self._out('/Subject '+self._textstring(self.subject))
1648        if hasattr(self,'author'):
1649            self._out('/Author '+self._textstring(self.author))
1650        if hasattr (self,'keywords'):
1651            self._out('/Keywords '+self._textstring(self.keywords))
1652        if hasattr(self,'creator'):
1653            self._out('/Creator '+self._textstring(self.creator))
1654        self._out('/CreationDate '+self._textstring('D:'+datetime.now().strftime('%Y%m%d%H
1656    def _putcatalog(self):
1657        self._out('/Type /Catalog')
1658        self._out('/Pages 1 0 R')
1659        if(self.zoom_mode=='fullpage'):
1660            self._out('/OpenAction [3 0 R /Fit]')
1661        elif(self.zoom_mode=='fullwidth'):
1662            self._out('/OpenAction [3 0 R /FitH null]')
1663        elif(self.zoom_mode=='real'):
1664            self._out('/OpenAction [3 0 R /XYZ null null 1]')
1665        elif(not isinstance(self.zoom_mode,basestring)):
1666            self._out(sprintf('/OpenAction [3 0 R /XYZ null null %s]',self.zoom_mode/100))
1667        if(self.layout_mode=='single'):
1668            self._out('/PageLayout /SinglePage')
1669        elif(self.layout_mode=='continuous'):
1670            self._out('/PageLayout /OneColumn')
1671        elif(self.layout_mode=='two'):
1672            self._out('/PageLayout /TwoColumnLeft')
1674    def _putheader(self):
1675        self._out('%PDF-'+self.pdf_version)
```

```
1677    def _puttrailer(self):
1678        self._out('/Size '+str(self.n+1))
1679        self._out('/Root '+str(self.n)+' 0 R')
1680        self._out('/Info '+str(self.n-1)+' 0 R')
1682    def _enddoc(self):
1683        self._putheader()
1684        self._putpages()
1685        self._putresources()
1686        #Info
1687        self._newobj()
1688        self._out('<<')
1689        self._putinfo()
1690        self._out('>>')
1691        self._out('endobj')
1692        #Catalog
1693        self._newobj()
1694        self._out('<<')
1695        self._putcatalog()
1696        self._out('>>')
1697        self._out('endobj')
1698        #Cross-ref
1699        o=len(self.buffer)
1700        self._out('xref')
1701        self._out('0 '+(str(self.n+1)))
1702        self._out('0000000000 65535 f ')
1703        for i in range(1,self.n+1):
1704            self._out(sprintf('%010d 00000 n ',self.offsets[i]))
1705        #Trailer
1706        self._out('trailer')
1707        self._out('<<')
1708        self._puttrailer()
1709        self._out('>>')
1710        self._out('startxref')
1711        self._out(o)
1712        self._out('%%EOF')
1713        self.state=3
```

```
1715    def _beginpage(self, orientation, format, same):
1716        self.page += 1
1717        self.pages[self.page] = {"content": ""}
1718        self.state = 2
1719        self.x = self.l_margin
1720        self.y = self.t_margin
1721        self.font_family = ''
1722        self.font_stretching = 100
1723        if not same:
1724            # Page format
1725            if format:
1726                # Change page format
1727                self.fw_pt, self.fh_pt = self.get_page_format(format, self.k)
1728            else:
1729                # Set to default format
1730                self.fw_pt = self.dw_pt
1731                self.fh_pt = self.dh_pt
1732            self.fw = self.fw_pt / self.k
1733            self.fh = self.fh_pt / self.k
1734            # Page orientation
1735            if not orientation:
1736                orientation = self.def_orientation
1737            else:
1738                orientation = orientation[0].upper()
1739            if orientation == 'P':
1740                self.w_pt = self.fw_pt
1741                self.h_pt = self.fh_pt
1742            else:
1743                self.w_pt = self.fh_pt
1744                self.h_pt = self.fw_pt
1745            self.w = self.w_pt / self.k
1746            self.h = self.h_pt / self.k
1747            self.cur_orientation = orientation
1748            self.page_break_trigger = self.h - self.b_margin
1749            self.cur_orientation = orientation
1750        self.pages[self.page]["w_pt"] = self.w_pt
1751        self.pages[self.page]["h_pt"] = self.h_pt
1753    def _endpage(self):
1754        #End of page contents
1755        self.state=1
```

```
    def _newobj(self):
        #Begin a new object
        self.n+=1
        self.offsets[self.n]=len(self.buffer)
        self._out(str(self.n)+' 0 obj')
    def _dounderline(self, x, y, txt):
        #Underline text
        up=self.current_font['up']
        ut=self.current_font['ut']
        w=self.get_string_width(txt, True)+self.ws*txt.count(' ')
        return sprintf('%.2f %.2f %.2f %.2f re f',x*self.k,(self.h-(y-up/1000.0*self.font_
    def load_resource(self, reason, filename):
        "Load external file"
        # by default loading from network is allowed for all images
        if reason == "image":
            if filename.startswith("http://") or filename.startswith("https://"):
                f = BytesIO(urlopen(filename).read())
            else:
                f = open(filename, "rb")
            return f
        else:
            self.error("Unknown resource loading reason \"%s\"" % reason)
```

```
1782    def _parsejpg(self, filename):
1783        # Extract info from a JPEG file
1784        f = None
1785        try:
1786            f = self.load_resource("image", filename)
1787            while True:
1788                markerHigh, markerLow = struct.unpack('BB', f.read(2))
1789                if markerHigh != 0xFF or markerLow < 0xC0:
1790                    raise SyntaxError('No JPEG marker found')
1791                elif markerLow == 0xDA: # SOS
1792                    raise SyntaxError('No JPEG SOF marker found')
1793                elif (markerLow == 0xC8 or # JPG
1794                      (markerLow >= 0xD0 and markerLow <= 0xD9) or # RSTx
1795                      (markerLow >= 0xF0 and markerLow <= 0xFD)): # JPGx
1796                    pass
1797                else:
1798                    dataSize, = struct.unpack('>H', f.read(2))
1799                    data = f.read(dataSize - 2) if dataSize > 2 else ''
1800                    if ((markerLow >= 0xC0 and markerLow <= 0xC3) or # SOF0 - SOF3
1801                        (markerLow >= 0xC5 and markerLow <= 0xC7) or # SOF4 - SOF7
1802                        (markerLow >= 0xC9 and markerLow <= 0xCB) or # SOF9 - SOF11
1803                        (markerLow >= 0xCD and markerLow <= 0xCF)): # SOF13 - SOF15
1804                        bpc, height, width, layers = struct.unpack_from('>BHHB', data)
1805                        colspace = 'DeviceRGB' if layers == 3 else ('DeviceCMYK' if layers
1806                        break
1807        except Exception:
1808            if f:
1809                f.close()
1810            self.error('Missing or incorrect image file: %s. error: %s' % (filename, str(e
1811
1812        with f:
1813            # Read whole file from the start
1814            f.seek(0)
1815            data = f.read()
1816        return {'w':width,'h':height,'cs':colspace,'bpc':bpc,'f':'DCTDecode','data':data}
```

```
1818     def _parsegif(self, filename):
1819         # Extract info from a GIF file (via PNG conversion)
1820         if Image is None:
1821             self.error('PIL is required for GIF support')
1822         try:
1823             im = Image.open(filename)
1824         except Exception:
1825             self.error('Missing or incorrect image file: %s. error: %s' % (filename, str(e
1826         else:
1827             # Use temporary file
1828             with tempfile.NamedTemporaryFile(delete=False, suffix=".png") as \
1829                     f:
1830                 tmp = f.name
1831             if "transparency" in im.info:
1832                 im.save(tmp, transparency = im.info['transparency'])
1833             else:
1834                 im.save(tmp)
1835             info = self._parsepng(tmp)
1836             os.unlink(tmp)
1837         return info
1839     def _parsepng(self, filename):
1840         #Extract info from a PNG file
1841         f = self.load_resource("image", filename)
1842         #Check signature
1843         magic = f.read(8).decode("latin1")
1844         signature = '\x89'+'PNG'+'\r'+'\n'+'\x1a'+'\n'
1845         if not PY3K: signature = signature.decode("latin1")
1846         if(magic!=signature):
1847             self.error('Not a PNG file: ' + filename)
1848         #Read header chunk
1849         f.read(4)
1850         chunk = f.read(4).decode("latin1")
1851         if(chunk!='IHDR'):
1852             self.error('Incorrect PNG file: ' + filename)
1853         w=self._freadint(f)
1854         h=self._freadint(f)
1855         bpc=ord(f.read(1))
1856         if(bpc>8):
1857             self.error('16-bit depth not supported: ' + filename)
1858         ct=ord(f.read(1))
1859         if(ct==0 or ct==4):
1860             colspace='DeviceGray'
1861         elif(ct==2 or ct==6):
1862             colspace='DeviceRGB'
```

```
        elif(ct==3):
            colspace='Indexed'
        else:
            self.error('Unknown color type: ' + filename)
        if(ord(f.read(1))!=0):
            self.error('Unknown compression method: ' + filename)
        if(ord(f.read(1))!=0):
            self.error('Unknown filter method: ' + filename)
        if(ord(f.read(1))!=0):
            self.error('Interlacing not supported: ' + filename)
        f.read(4)
        dp='/Predictor 15 /Colors '
        if colspace == 'DeviceRGB':
            dp+='3'
        else:
            dp+='1'
        dp+=' /BitsPerComponent '+str(bpc)+' /Columns '+str(w)+''
        #Scan chunks looking for palette, transparency and image data
        pal=''
        trns=''
        data=bytes() if PY3K else str()
        n=1
        while n != None:
            n=self._freadint(f)
            type=f.read(4).decode("latin1")
            if(type=='PLTE'):
                #Read palette
                pal=f.read(n)
                f.read(4)
            elif(type=='tRNS'):
                #Read transparency info
                t=f.read(n)
                if(ct==0):
                    trns=[ord(substr(t,1,1)),]
                elif(ct==2):
                    trns=[ord(substr(t,1,1)),ord(substr(t,3,1)),ord(substr(t,5,1))]
                else:
                    pos=t.find('\x00'.encode("latin1"))
                    if(pos!=-1):
                        trns=[pos,]
                f.read(4)
            elif(type=='IDAT'):
                #Read image data block
                data+=f.read(n)
                f.read(4)
```

```
            elif(type=='IEND'):
                break
            else:
                f.read(n+4)
        if(colspace=='Indexed' and not pal):
            self.error('Missing palette in ' + filename)
        f.close()
        info = {'w':w,'h':h,'cs':colspace,'bpc':bpc,'f':'FlateDecode','dp':dp,'pal':pal,'t
        if(ct>=4):
            # Extract alpha channel
            data = zlib.decompress(data)
            color = b('')
            alpha = b('')
            if(ct==4):
                # Gray image
                length = 2*w
                for i in range(h):
                    pos = (1+length)*i
                    color += b(data[pos])
                    alpha += b(data[pos])
                    line = substr(data, pos+1, length)
                    re_c = re.compile('(.).'.encode("ascii"), flags=re.DOTALL)
                    re_a = re.compile('.(.)'.encode("ascii"), flags=re.DOTALL)
                    color += re_c.sub(lambda m: m.group(1), line)
                    alpha += re_a.sub(lambda m: m.group(1), line)
            else:
                # RGB image
                length = 4*w
                for i in range(h):
                    pos = (1+length)*i
                    color += b(data[pos])
                    alpha += b(data[pos])
                    line = substr(data, pos+1, length)
                    re_c = re.compile('(...).'.encode("ascii"), flags=re.DOTALL)
                    re_a = re.compile('...(.)'.encode("ascii"), flags=re.DOTALL)
                    color += re_c.sub(lambda m: m.group(1), line)
                    alpha += re_a.sub(lambda m: m.group(1), line)
            del data
            data = zlib.compress(color)
            info['smask'] = zlib.compress(alpha)
            if (self.pdf_version < '1.4'):
                self.pdf_version = '1.4'
        info['data'] = data
        return info
```

```
    def _freadint(self, f):
        #Read a 4-byte integer from file
        try:
            return struct.unpack('>I', f.read(4))[0]
        except:
            return None
    def _textstring(self, s):
        #Format a text string
        return '('+self._escape(s)+')'
    def _escape(self, s):
        #Add \ before \, ( and )
        return s.replace('\\','\\\\').replace(')','\\)').replace('(','\\(').replace('\r','
    def _putstream(self, s):
        self._out('stream')
        self._out(s)
        self._out('endstream')
    def _out(self, s):
        #Add a line to the document
        if PY3K and isinstance(s, bytes):
            # manage binary data as latin1 until PEP461-like function is implemented
            s = s.decode("latin1")
        elif not PY3K and isinstance(s, unicode):
            s = s.encode("latin1")    # default encoding (font name and similar)
        elif not isinstance(s, basestring):
            s = str(s)
        if(self.state == 2):
            self.pages[self.page]["content"] += (s + "\n")
        else:
            self.buffer += (s + "\n")
    def interleaved2of5(self, txt, x, y, w=1.0, h=10.0):
        "Barcode I2of5 (numeric), adds a 0 if odd lenght"
        narrow = w / 3.0
        wide = w

        # wide/narrow codes for the digits
        bar_char={'0': 'nnwwn', '1': 'wnnnw', '2': 'nwnnw', '3': 'wwnnn',
                  '4': 'nnwnw', '5': 'wnwnn', '6': 'nwwnn', '7': 'nnnww',
                  '8': 'wnnwn', '9': 'nwnwn', 'A': 'nn', 'Z': 'wn'}

        self.set_fill_color(0)
        code = txt
        # add leading zero if code-length is odd
        if len(code) % 2 != 0:
```

```
2002            code = '0' + code
2003
2004        # add start and stop codes
2005        code = 'AA' + code.lower() + 'ZA'
2006
2007        for i in range(0, len(code), 2):
2008            # choose next pair of digits
2009            char_bar = code[i]
2010            char_space = code[i+1]
2011            # check whether it is a valid digit
2012            if not char_bar in bar_char.keys():
2013                raise RuntimeError ('Char "%s" invalid for I25: ' % char_bar)
2014            if not char_space in bar_char.keys():
2015                raise RuntimeError ('Char "%s" invalid for I25: ' % char_space)
2016
2017            # create a wide/narrow-seq (first digit=bars, second digit=spaces)
2018            seq = ''
2019            for s in range(0, len(bar_char[char_bar])):
2020                seq += bar_char[char_bar][s] + bar_char[char_space][s]
2021
2022            for bar in range(0, len(seq)):
2023                # set line_width depending on value
2024                if seq[bar] == 'n':
2025                    line_width = narrow
2026                else:
2027                    line_width = wide
2028
2029                # draw every second value, the other is represented by space
2030                if bar % 2 == 0:
2031                    self.rect(x, y, line_width, h, 'F')
2032
2033                x += line_width
```

```
    def code39(self, txt, x, y, w=1.5, h=5.0):
        """Barcode 3of9"""
        dim = {'w': w, 'n': w/3.}
        chars = {
            '0': 'nnnwwnwnn', '1': 'wnnwnnnnw', '2': 'nnwwnnnnw',
            '3': 'wnwwnnnnn', '4': 'nnnwwnnnw', '5': 'wnnwwnnnn',
            '6': 'nnwwwnnnn', '7': 'nnnwnnwnw', '8': 'wnnwnnwnn',
            '9': 'nnwwnnwnn', 'A': 'wnnnnwnnw', 'B': 'nnwnnwnnw',
            'C': 'wnwnnwnnn', 'D': 'nnnnwwnnw', 'E': 'wnnnwwnnn',
            'F': 'nnwnwwnnn', 'G': 'nnnnnwwnw', 'H': 'wnnnnwwnn',
            'I': 'nnwnnwwnn', 'J': 'nnnnwwwnn', 'K': 'wnnnnnnww',
            'L': 'nnwnnnnww', 'M': 'wnwnnnnwn', 'N': 'nnnnwnnww',
            'O': 'wnnnwnnwn', 'P': 'nnwnwnnwn', 'Q': 'nnnnnnwww',
            'R': 'wnnnnnwwn', 'S': 'nnwnnnwwn', 'T': 'nnnnwnwwn',
            'U': 'wnwnnnnnw', 'V': 'nwwnnnnnw', 'W': 'wnwnnnnnn',
            'X': 'nwnnwnnnw', 'Y': 'wnwnnwnnn', 'Z': 'nwwnwnnnn',
            '-': 'nwnnnnwnw', '.': 'wnwnnnnwn', ' ': 'nwwnnnwnn',
            '*': 'nwnnwnwnn', '$': 'nwnwnwnnn', '/': 'nwnwnnnwn',
            '+': 'nwnnnwnwn', '%': 'nnnwnwnwn',
        }
        self.set_fill_color(0)
        for c in txt.upper():
            if c not in chars:
                raise RuntimeError('Invalid char "%s" for Code39' % c)
            for i, d in enumerate(chars[c]):
                if i % 2 == 0:
                    self.rect(x, y, dim[d], h, 'F')
                x += dim[d]
            x += dim['n']
```