```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),
}
```

Do we need this import? (handwritten annotation pointing to line 22)

Check for null checks in the code (handwritten annotation pointing to lines 33-34)

```python
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',
```

(handwritten note pointing to line 44:) Check this function

(handwritten note:) Some Random Comments here.

```python
 92                 'courierI': 'Courier-Oblique', 'courierBI': 'Courier-BoldOblique',
 93                 'helvetica': 'Helvetica', 'helveticaB': 'Helvetica-Bold',
 94                 'helveticaI': 'Helvetica-Oblique',
 95                 'helveticaBI': 'Helvetica-BoldOblique',
 96                 'times': 'Times-Roman', 'timesB': 'Times-Bold',
 97                 'timesI': 'Times-Italic', 'timesBI': 'Times-BoldItalic',
 98                 'symbol': 'Symbol', 'zapfdingbats': 'ZapfDingbats'}
 99             self.core_fonts_encoding = "latin-1"
100             # Scale factor
101             if unit == "pt":
102                 self.k = 1
103             elif unit == "mm":
104                 self.k = 72 / 25.4
105             elif unit == "cm":
106                 self.k = 72 / 2.54
107             elif unit == 'in':
108                 self.k = 72.
109             else:
110                 self.error("Incorrect unit: " + unit)
111             # Page format
112             self.fw_pt, self.fh_pt = self.get_page_format(format, self.k)
113             self.dw_pt = self.fw_pt
114             self.dh_pt = self.fh_pt
115             self.fw = self.fw_pt / self.k
116             self.fh = self.fh_pt / self.k
117             # Page orientation
118             orientation = orientation.lower()
119             if orientation in ('p', 'portrait'):
120                 self.def_orientation = 'P'
121                 self.w_pt = self.fw_pt
122                 self.h_pt = self.fh_pt
123             elif orientation in ('l', 'landscape'):
124                 self.def_orientation = 'L'
125                 self.w_pt = self.fh_pt
126                 self.h_pt = self.fw_pt
127             else:
128                 self.error('Incorrect orientation: ' + orientation)
129             self.cur_orientation = self.def_orientation
130             self.w = self.w_pt / self.k
131             self.h = self.h_pt / self.k
132             # Page margins (1 cm)
133             margin = 28.35 / self.k
134             self.set_margins(margin, margin)
135             # Interior cell margin (1 mm)
136             self.c_margin = margin / 10.0
```

```python
137            # line width (0.2 mm)
138            self.line_width = .567 / self.k
139            # Automatic page break
140            self.set_auto_page_break(1, 2 * margin)
141            # Full width display mode
142            self.set_display_mode('fullwidth')
143            # Enable compression
144            self.set_compression(1)
145            # Set default PDF version number
146            self.pdf_version = '1.3'

149        def get_page_format(format, k):
150            "Return scale factor, page w and h size in points"
151            if isinstance(format, basestring):
152                format = format.lower()
153                if format in PAGE_FORMATS:
154                    return PAGE_FORMATS[format]
155                else:
156                    raise RuntimeError("Unknown page format: " + format)
157            else:
158                return (format[0] * k, format[1] * k)

160        def check_page(fn):
161            "Decorator to protect drawing methods"
162            @wraps(fn)
163            def wrapper(self, *args, **kwargs):
164                if not self.page and not kwargs.get('split_only'):
165                    self.error("No page open, you need to call add_page() first")
166                else:
167                    return fn(self, *args, **kwargs)
168            return wrapper

170        def set_margins(self, left,top,right=-1):
171            "Set left, top and right margins"
172            self.l_margin=left
173            self.t_margin=top
174            if(right==-1):
175                right=left
176            self.r_margin=right

178        def set_left_margin(self, margin):
179            "Set left margin"
180            self.l_margin=margin
181            if(self.page>0 and self.x<margin):
182                self.x=margin
```

```python
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
```

```python
233        def set_creator(self, creator):
234            "Creator of document"
235            self.creator=creator

237        def set_doc_option(self, opt, value):
238            "Set document option"
239            if opt == "core_fonts_encoding":
240                self.core_fonts_encoding = value
241            else:
242                self.error("Unknown document option \"%s\"" % str(opt))

244        def alias_nb_pages(self, alias='{nb}'):
245            "Define an alias for total number of pages"
246            self.str_alias_nb_pages=alias
247            return alias

249        def error(self, msg):
250            "Fatal error"
251            raise RuntimeError('FPDF error: '+msg)

253        def open(self):
254            "Begin document"
255            self.state=1

257        def close(self):
258            "Terminate document"
259            if(self.state==3):
260                return
261            if(self.page==0):
262                self.add_page()
263            #Page footer
264            self.in_footer=1
265            self.footer()
266            self.in_footer=0
267            #close page
268            self._endpage()
269            #close document
270            self._enddoc()

272        def add_page(self, orientation = '', format = '', same = False):
273            "Start a new page, if same page format will be same as previous"
274            if(self.state==0):
275                self.open()
276            family=self.font_family
277            if self.underline:
278                style = self.font_style + 'U'
279            else:
280                style = self.font_style
281            size=self.font_size_pt
```

```
282            lw=self.line_width
283            dc=self.draw_color
284            fc=self.fill_color
285            tc=self.text_color
286            cf=self.color_flag
287            stretching=self.font_stretching
288        if(self.page>0):
289            #Page footer
290            self.in_footer=1
291            self.footer()
292            self.in_footer=0
293            #close page
294            self._endpage()
295        #Start new page
296        self._beginpage(orientation, format, same)
297        #Set line cap style to square
298        self._out('2 J')
299        #Set line width
300        self.line_width=lw
301        self._out(sprintf('%.2f w',lw*self.k))
302        #Set font
303        if(family):
304            self.set_font(family,style,size)
305        #Set colors
306        self.draw_color=dc
307        if(dc!='0 G'):
308            self._out(dc)
309        self.fill_color=fc
310        if(fc!='0 g'):
311            self._out(fc)
312        self.text_color=tc
313        self.color_flag=cf
314        #Page header
315        self.header()
316        #Restore line width
317        if(self.line_width!=lw):
318            self.line_width=lw
319            self._out(sprintf('%.2f w',lw*self.k))
320        #Restore font
321        if(family):
322            self.set_font(family,style,size)
323        #Restore colors
324        if(self.draw_color!=dc):
325            self.draw_color=dc
326            self._out(dc)
```

```python
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)
```

```python
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->CurrentFc
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)
```

```python
    def dashed_line(self, x1,y1,x2,y2, dash_length=1, space_length=1):
        """Draw a dashed line. Same interface as line() except:
            - dash_length: Length of the dash
            - space_length: Length of the space between dashes"""
        self._set_dash(dash_length, space_length)
        self.line(x1, y1, x2, y2)
        self._set_dash()

    def rect(self, x,y,w,h,style=''):
        "Draw a rectangle"
        if(style=='F'):
            op='f'
        elif(style=='FD' or style=='DF'):
            op='B'
        else:
            op='S'
        self._out(sprintf('%.2f %.2f %.2f %.2f re %s',x*self.k,(self.h-y)*self.k,w*self.k,
```

```python
439      def ellipse(self, x,y,w,h,style=''):
440          "Draw a ellipse"
441          if(style=='F'):
442              op='f'
443          elif(style=='FD' or style=='DF'):
444              op='B'
445          else:
446              op='S'
447
448          cx = x + w/2.0
449          cy = y + h/2.0
450          rx = w/2.0
451          ry = h/2.0
452
453          lx = 4.0/3.0*(math.sqrt(2)-1)*rx
454          ly = 4.0/3.0*(math.sqrt(2)-1)*ry
455
456          self._out(sprintf('%.2f %.2f m %.2f %.2f %.2f %.2f %.2f %.2f c',
457              (cx+rx)*self.k, (self.h-cy)*self.k,
458              (cx+rx)*self.k, (self.h-(cy-ly))*self.k,
459              (cx+lx)*self.k, (self.h-(cy-ry))*self.k,
460              cx*self.k, (self.h-(cy-ry))*self.k))
461          self._out(sprintf('%.2f %.2f %.2f %.2f %.2f %.2f c',
462              (cx-lx)*self.k, (self.h-(cy-ry))*self.k,
463              (cx-rx)*self.k, (self.h-(cy-ly))*self.k,
464              (cx-rx)*self.k, (self.h-cy)*self.k))
465          self._out(sprintf('%.2f %.2f %.2f %.2f %.2f %.2f c',
466              (cx-rx)*self.k, (self.h-(cy+ly))*self.k,
467              (cx-lx)*self.k, (self.h-(cy+ry))*self.k,
468              cx*self.k, (self.h-(cy+ry))*self.k))
469          self._out(sprintf('%.2f %.2f %.2f %.2f %.2f %.2f c %s',
470              (cx+lx)*self.k, (self.h-(cy+ry))*self.k,
471              (cx+rx)*self.k, (self.h-(cy+ly))*self.k,
472              (cx+rx)*self.k, (self.h-cy)*self.k,
473              op))
475      def add_font(self, family, style='', fname='', uni=False):
476          "Add a TrueType or Type1 font"
477          family = family.lower()
478          if (fname == ''):
479              fname = family.replace(' ','') + style.lower() + '.pkl'
480          if (family == 'arial'):
481              family = 'helvetica'
482          style = style.upper()
483          if (style == 'IB'):
```

```python
                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 = {
```

```python
                            '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):
```

```
574                          d = i
575                              break
576                      if (d == 0):
577                          d = nb + 1
578                          self.diffs[d] = diff
579                      self.fonts[fontkey]['diff'] = d
580              filename = font_dict.get('filename')
581              if (filename):
582                  if (font_dict['type'] == 'TrueType'):
583                      originalsize = font_dict['originalsize']
584                      self.font_files[filename]={'length1': originalsize}
585                  else:
586                      self.font_files[filename]={'length1': font_dict['size1'],
587                                                 'length2': font_dict['size2']}

589      def set_font(self, family,style='',size=0):
590          "Select a font; size given in points"
591          family=family.lower()
592          if(family==''):
593              family=self.font_family
594          if(family=='arial'):
595              family='helvetica'
596          elif(family=='symbol' or family=='zapfdingbats'):
597              style=''
598          style=style.upper()
599          if('U' in style):
600              self.underline=1
601              style=style.replace('U','')
602          else:
603              self.underline=0
604          if(style=='IB'):
605              style='BI'
606          if(size==0):
607              size=self.font_size_pt
608          #Test if font is already selected
609          if(self.font_family==family and self.font_style==style and self.font_size_pt==size
610              return
611          #Test if used for the first time
612          fontkey=family+style
613          if fontkey not in self.fonts:
614              #Check if one of the standard fonts
615              if fontkey in self.core_fonts:
616                  if fontkey not in fpdf_charwidths:
617                      #Load metric file
618                      name=os.path.join(FPDF_FONT_DIR,family)
```

```python
                    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
```

```python
662        def set_link(self, link,y=0,page=-1):
663            "Set destination of internal link"
664            if(y==-1):
665                y=self.y
666            if(page==-1):
667                page=self.page
668            self.links[link]=[page,y]

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

677        def text(self, x, y, txt=''):
678            "Output a string"
679            txt = self.normalize_text(txt)
680            if (self.unifontsubset):
681                txt2 = self._escape(UTF8ToUTF16BE(txt, False))
682                for uni in UTF8StringToArray(txt):
683                    self.current_font['subset'].append(uni)
684            else:
685                txt2 = self._escape(txt)
686            s=sprintf('BT %.2f %.2f Td (%s) Tj ET',x*self.k,(self.h-y)*self.k, txt2)
687            if(self.underline and txt!=''):
688                s+=' '+self._dounderline(x,y,txt)
689            if(self.color_flag):
690                s='q '+self.text_color+' '+s+' Q'
691            self._out(s)

694        def rotate(self, angle, x=None, y=None):
695            if x is None:
696                x = self.x
697            if y is None:
698                y = self.y;
699            if self.angle!=0:
700                self._out('Q')
701            self.angle = angle
702            if angle!=0:
703                angle *= math.pi/180;
704                c = math.cos(angle);
705                s = math.sin(angle);
706                cx = x*self.k;
707                cy = (self.h-y)*self.k
708                s = sprintf('q %.5F %.5F %.5F %.5F %.2F %.2F cm 1 0 0 1 %.2F %.2F cm',c,s,-s,c
709                self._out(s)
```

```python
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'):
```

```python
                    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
```

```python
802                self.y+=h
803                if(ln==1):
804                    self.x=self.l_margin
805            else:
806                self.x+=w

809    def multi_cell(self, w, h, txt='', border=0, align='J', fill=0, split_only=False):
810        "Output text with automatic or explicit line breaks"
811        txt = self.normalize_text(txt)
812        ret = [] # if split_only = True, returns splited text cells
813        cw=self.current_font['cw']
814        if(w==0):
815            w=self.w-self.r_margin-self.x
816        wmax=(w-2*self.c_margin)*1000.0/self.font_size
817        s=txt.replace("\r",'')
818        nb=len(s)
819        if(nb>0 and s[nb-1]=="\n"):
820            nb-=1
821        b=0
822        if(border):
823            if(border==1):
824                border='LTRB'
825                b='LRT'
826                b2='LR'
827            else:
828                b2=''
829                if('L' in border):
830                    b2+='L'
831                if('R' in border):
832                    b2+='R'
833                if ('T' in border):
834                    b=b2+'T'
835                else:
836                    b=b2
837        sep=-1
838        i=0
839        j=0
840        l=0
841        ns=0
842        nl=1
843        while(i<nb):
844            #Get next character
845            c=s[i]
846            if(c=="\n"):
847                #Explicit line break
```

```python
            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]
```

```python
            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
```

```python
984                else:
985                    i+=1
986            #Last chunk
987            if(i!=j):
988                self.cell(l/1000.0*self.font_size,h,substr(s,j),0,0,'',0,link)
991    def image(self, name, x=None, y=None, w=0,h=0,type='',link='', is_mask=False, mask_ima
992        "Put an image on the page"
993        if not name in self.images:
994            #First use of image, get info
995            if(type==''):
996                pos=name.rfind('.')
997                if(not pos):
998                    self.error('image file has no extension and no type was specified: '+n
999                type=substr(name,pos+1)
1000            type=type.lower()
1001            if(type=='jpg' or type=='jpeg'):
1002                info=self._parsejpg(name)
1003            elif(type=='png'):
1004                info=self._parsepng(name)
1005            else:
1006                #Allow for additional formats
1007                #maybe the image is not showing the correct extension,
1008                #but the header is OK,
1009                succeed_parsing = False
1010                #try all the parsing functions
1011                parsing_functions = [self._parsejpg,self._parsepng,self._parsegif]
1012                for pf in parsing_functions:
1013                    try:
1014                        info = pf(name)
1015                        succeed_parsing = True
1016                        break;
1017                    except:
1018                        pass
1019                #last resource
1020                if not succeed_parsing:
1021                    mtd='_parse'+type
1022                    if not hasattr(self,mtd):
1023                        self.error('Unsupported image type: '+type)
1024                    info=getattr(self, mtd)(name)
1025            mtd='_parse'+type
1026            if not hasattr(self,mtd):
1027                self.error('Unsupported image type: '+type)
1028            info=getattr(self, mtd)(name)
1029        info['i']=len(self.images)+1
```

```
1030                    # is_mask and mask_image
1031                    if is_mask and info['cs'] != 'DeviceGray':
1032                        self.error('Mask must be a gray scale image')
1033                    if mask_image:
1034                        info['masked'] = mask_image
1035                    self.images[name]=info
1036                else:
1037                    info=self.images[name]
1038                #Automatic width and height calculation if needed
1039                if(w==0 and h==0):
1040                    #Put image at 72 dpi
1041                    w=info['w']/self.k
1042                    h=info['h']/self.k
1043                elif(w==0):
1044                    w=h*info['w']/info['h']
1045                elif(h==0):
1046                    h=w*info['h']/info['w']
1047                # Flowing mode
1048                if y is None:
1049                    if (self.y + h > self.page_break_trigger and not self.in_footer and self.accep
1050                        #Automatic page break
1051                        x = self.x
1052                        self.add_page(same = True)
1053                        self.x = x
1054                    y = self.y
1055                    self.y += h
1056                if x is None:
1057                    x = self.x
1058                if not is_mask:
1059                    self._out(sprintf('q %.2f 0 0 %.2f %.2f %.2f cm /I%d Do Q',w*self.k,h*self.k,x
1060                if(link):
1061                    self.link(x,y,w,h,link)
1062
1063                return info
1066            def ln(self, h=''):
1067                "Line Feed; default value is last cell height"
1068                self.x=self.l_margin
1069                if(isinstance(h, basestring)):
1070                    self.y+=self.lasth
1071                else:
1072                    self.y+=h
1074            def get_x(self):
1075                "Get x position"
1076                return self.x
```

```python
1078    def set_x(self, x):
1079        "Set x position"
1080        if(x>=0):
1081            self.x=x
1082        else:
1083            self.x=self.w+x

1085    def get_y(self):
1086        "Get y position"
1087        return self.y

1089    def set_y(self, y):
1090        "Set y position and reset x"
1091        self.x=self.l_margin
1092        if(y>=0):
1093            self.y=y
1094        else:
1095            self.y=self.h+y

1097    def set_xy(self, x,y):
1098        "Set x and y positions"
1099        self.set_y(y)
1100        self.set_x(x)
```

```python
def output(self, name='',dest=''):
    """Output PDF to some destination

    By default the PDF is written to sys.stdout. If a name is given, the
    PDF is written to a new file. If dest='S' is given, the PDF data is
    returned as a byte string."""

    #Finish document if necessary
    if(self.state<3):
        self.close()
    dest=dest.upper()
    if(dest==''):
        if(name==''):
            dest='I'
        else:
            dest='F'
    if PY3K:
        # manage binary data as latin1 until PEP461 or similar is implemented
        buffer = self.buffer.encode("latin1")
    else:
        buffer = self.buffer
    if dest in ('I', 'D'):
        # Python < 3 writes byte data transparently without "buffer"
        stdout = getattr(sys.stdout, 'buffer', sys.stdout)
        stdout.write(buffer)
    elif dest=='F':
        #Save to local file
        with open(name,'wb') as f:
            f.write(buffer)
    elif dest=='S':
        #Return as a byte string
        return buffer
    else:
        self.error('Incorrect output destination: '+dest)
```

```python
1137        def normalize_text(self, txt):
1138            "Check that text input is in the correct format/encoding"
1139            # - for TTF unicode fonts: unicode object (utf8 encoding)
1140            # - for built-in fonts: string instances (encoding: latin-1, cp1252)
1141            if not PY3K:
1142                if self.unifontsubset and isinstance(txt, str):
1143                    return txt.decode("utf-8")
1144                elif not self.unifontsubset and isinstance(txt, unicode):
1145                    return txt.encode(self.core_fonts_encoding)
1146            else:
1147                if not self.unifontsubset and self.core_fonts_encoding:
1148                    return txt.encode(self.core_fonts_encoding).decode("latin-1")
1149            return txt

1151        def _dochecks(self):
1152            if(sprintf('%.1f',1.0)!='1.0'):
1153                import locale
1154                locale.setlocale(locale.LC_NUMERIC,'C')

1156        def _getfontpath(self):
1157            return FPDF_FONT_DIR+'/'

1159        def _putpages(self):
1160            nb = self.page
1161            if hasattr(self, 'str_alias_nb_pages'):
1162                # Replace number of pages in fonts using subsets (unicode)
1163                alias = UTF8ToUTF16BE(self.str_alias_nb_pages, False)
1164                r = UTF8ToUTF16BE(str(nb), False)
1165                for n in range(1, nb + 1):
1166                    self.pages[n]["content"] = \
1167                        self.pages[n]["content"].replace(alias, r)
1168                # Now repeat for no pages in non-subset fonts
1169                for n in range(1,nb + 1):
1170                    self.pages[n]["content"] = \
1171                        self.pages[n]["content"].replace(self.str_alias_nb_pages,
1172                            str(nb))
1173            if self.def_orientation == 'P':
1174                dw_pt = self.dw_pt
1175                dh_pt = self.dh_pt
1176            else:
1177                dw_pt = self.dh_pt
1178                dh_pt = self.dw_pt
1179            if self.compress:
1180                filter = '/Filter /FlateDecode '
1181            else:
1182                filter = ''
1183            for n in range(1, nb + 1):
```

```python
            # 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)
```

```python
                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)
```

```python
1274                        self._out('endobj')
1275            flist = [(x[1]["i"],x[0],x[1]) for x in self.fonts.items()]
1276            flist.sort()
1277            for idx,k,font in flist:
1278                #Font objects
1279                self.fonts[k]['n']=self.n+1
1280                type=font['type']
1281                name=font['name']
1282                if(type=='core'):
1283                    #Standard font
1284                    self._newobj()
1285                    self._out('<</Type /Font')
1286                    self._out('/BaseFont /'+name)
1287                    self._out('/Subtype /Type1')
1288                    if(name!='Symbol' and name!='ZapfDingbats'):
1289                        self._out('/Encoding /WinAnsiEncoding')
1290                    self._out('>>')
1291                    self._out('endobj')
1292                elif(type=='Type1' or type=='TrueType'):
1293                    #Additional Type1 or TrueType font
1294                    self._newobj()
1295                    self._out('<</Type /Font')
1296                    self._out('/BaseFont /'+name)
1297                    self._out('/Subtype /'+type)
1298                    self._out('/FirstChar 32 /LastChar 255')
1299                    self._out('/Widths '+str(self.n+1)+' 0 R')
1300                    self._out('/FontDescriptor '+str(self.n+2)+' 0 R')
1301                    if(font['enc']):
1302                        if('diff' in font):
1303                            self._out('/Encoding '+str(nf+font['diff'])+' 0 R')
1304                        else:
1305                            self._out('/Encoding /WinAnsiEncoding')
1306                    self._out('>>')
1307                    self._out('endobj')
1308                    #Widths
1309                    self._newobj()
1310                    cw=font['cw']
1311                    s='['
1312                    for i in range(32,256):
1313                        # Get doesn't raise exception; returns 0 instead of None if not set
1314                        s+=str(cw.get(chr(i)) or 0)+' '
1315                    self._out(s+']')
1316                    self._out('endobj')
1317                    #Descriptor
1318                    self._newobj()
```

```python
                    s='<</Type /FontDescriptor /FontName /'+name
                    for k in ('Ascent', 'Descent', 'CapHeight', 'Flags', 'FontBBox', 'ItalicAn
                        s += ' /%s %s' % (k, font['desc'][k])
                    filename=font['file']
                    if(filename):
                        s+=' /FontFile'
                        if type!='Type1':
                            s+='2'
                        s+=' '+str(self.font_files[filename]['n'])+' 0 R'
                    self._out(s+'>>')
                    self._out('endobj')
                elif (type == 'TTF'):
                    self.fonts[k]['n'] = self.n + 1
                    ttf = TTFontFile()
                    fontname = 'MPDFAA' + '+' + font['name']
                    subset = font['subset']
                    del subset[0]
                    ttfontstream = ttf.makeSubset(font['ttffile'], subset)
                    ttfontsize = len(ttfontstream)
                    fontstream = zlib.compress(ttfontstream)
                    codeToGlyph = ttf.codeToGlyph
                    ##del codeToGlyph[0]
                    # Type0 Font
                    # A composite font - a font composed of other fonts, organized hierarchica
                    self._newobj()
                    self._out('<</Type /Font');
                    self._out('/Subtype /Type0');
                    self._out('/BaseFont /' + fontname + '');
                    self._out('/Encoding /Identity-H');
                    self._out('/DescendantFonts [' + str(self.n + 1) + ' 0 R]')
                    self._out('/ToUnicode ' + str(self.n + 2) + ' 0 R')
                    self._out('>>')
                    self._out('endobj')

                    # CIDFontType2
                    # A CIDFont whose glyph descriptions are based on TrueType font technology
                    self._newobj()
                    self._out('<</Type /Font')
                    self._out('/Subtype /CIDFontType2')
                    self._out('/BaseFont /' + fontname + '')
                    self._out('/CIDSystemInfo ' + str(self.n + 2) + ' 0 R')
                    self._out('/FontDescriptor ' + str(self.n + 3) + ' 0 R')
                    if (font['desc'].get('MissingWidth')):
                        self._out('/DW %d' % font['desc']['MissingWidth'])
                    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'):
```

```python
                        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'
```

```python
        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'])):
```

```python
                    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 = []
```

```python
        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])+' '
```

```python
                    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('>>')
```

```python
     def _putresources(self):
         self._putfonts()
         self._putimages()
         #Resource dictionary
         self.offsets[2]=len(self.buffer)
         self._out('2 0 obj')
         self._out('<<')
         self._putresourcedict()
         self._out('>>')
         self._out('endobj')

     def _putinfo(self):
         self._out('/Producer '+self._textstring('PyFPDF '+FPDF_VERSION+' http://pyfpdf.goc
         if hasattr(self,'title'):
             self._out('/Title '+self._textstring(self.title))
         if hasattr(self,'subject'):
             self._out('/Subject '+self._textstring(self.subject))
         if hasattr(self,'author'):
             self._out('/Author '+self._textstring(self.author))
         if hasattr (self,'keywords'):
             self._out('/Keywords '+self._textstring(self.keywords))
         if hasattr(self,'creator'):
             self._out('/Creator '+self._textstring(self.creator))
         self._out('/CreationDate '+self._textstring('D:'+datetime.now().strftime('%Y%m%d%H

     def _putcatalog(self):
         self._out('/Type /Catalog')
         self._out('/Pages 1 0 R')
         if(self.zoom_mode=='fullpage'):
             self._out('/OpenAction [3 0 R /Fit]')
         elif(self.zoom_mode=='fullwidth'):
             self._out('/OpenAction [3 0 R /FitH null]')
         elif(self.zoom_mode=='real'):
             self._out('/OpenAction [3 0 R /XYZ null null 1]')
         elif(not isinstance(self.zoom_mode,basestring)):
             self._out(sprintf('/OpenAction [3 0 R /XYZ null null %s]',self.zoom_mode/100))
         if(self.layout_mode=='single'):
             self._out('/PageLayout /SinglePage')
         elif(self.layout_mode=='continuous'):
             self._out('/PageLayout /OneColumn')
         elif(self.layout_mode=='two'):
             self._out('/PageLayout /TwoColumnLeft')

     def _putheader(self):
         self._out('%PDF-'+self.pdf_version)
```

```python
def _puttrailer(self):
    self._out('/Size '+str(self.n+1))
    self._out('/Root '+str(self.n)+' 0 R')
    self._out('/Info '+str(self.n-1)+' 0 R')

def _enddoc(self):
    self._putheader()
    self._putpages()
    self._putresources()
    #Info
    self._newobj()
    self._out('<<')
    self._putinfo()
    self._out('>>')
    self._out('endobj')
    #Catalog
    self._newobj()
    self._out('<<')
    self._putcatalog()
    self._out('>>')
    self._out('endobj')
    #Cross-ref
    o=len(self.buffer)
    self._out('xref')
    self._out('0 '+(str(self.n+1)))
    self._out('0000000000 65535 f ')
    for i in range(1,self.n+1):
        self._out(sprintf('%010d 00000 n ',self.offsets[i]))
    #Trailer
    self._out('trailer')
    self._out('<<')
    self._puttrailer()
    self._out('>>')
    self._out('startxref')
    self._out(o)
    self._out('%%EOF')
    self.state=3
```

```python
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
```

```python
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)
```

```python
def _parsejpg(self, filename):
    # Extract info from a JPEG file
    f = None
    try:
        f = self.load_resource("image", filename)
        while True:
            markerHigh, markerLow = struct.unpack('BB', f.read(2))
            if markerHigh != 0xFF or markerLow < 0xC0:
                raise SyntaxError('No JPEG marker found')
            elif markerLow == 0xDA: # SOS
                raise SyntaxError('No JPEG SOF marker found')
            elif (markerLow == 0xC8 or # JPG
                  (markerLow >= 0xD0 and markerLow <= 0xD9) or # RSTx
                  (markerLow >= 0xF0 and markerLow <= 0xFD)): # JPGx
                pass
            else:
                dataSize, = struct.unpack('>H', f.read(2))
                data = f.read(dataSize - 2) if dataSize > 2 else ''
                if ((markerLow >= 0xC0 and markerLow <= 0xC3) or # SOF0 - SOF3
                    (markerLow >= 0xC5 and markerLow <= 0xC7) or # SOF4 - SOF7
                    (markerLow >= 0xC9 and markerLow <= 0xCB) or # SOF9 - SOF11
                    (markerLow >= 0xCD and markerLow <= 0xCF)): # SOF13 - SOF15
                    bpc, height, width, layers = struct.unpack_from('>BHHB', data)
                    colspace = 'DeviceRGB' if layers == 3 else ('DeviceCMYK' if layers
                    break
    except Exception:
        if f:
            f.close()
        self.error('Missing or incorrect image file: %s. error: %s' % (filename, str(e

    with f:
        # Read whole file from the start
        f.seek(0)
        data = f.read()
    return {'w':width,'h':height,'cs':colspace,'bpc':bpc,'f':'DCTDecode','data':data}
```

```python
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'
```

```python
            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)
```

```python
                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
```

```python
    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:
```

```python
            code = '0' + code

        # add start and stop codes
        code = 'AA' + code.lower() + 'ZA'

        for i in range(0, len(code), 2):
            # choose next pair of digits
            char_bar = code[i]
            char_space = code[i+1]
            # check whether it is a valid digit
            if not char_bar in bar_char.keys():
                raise RuntimeError ('Char "%s" invalid for I25: ' % char_bar)
            if not char_space in bar_char.keys():
                raise RuntimeError ('Char "%s" invalid for I25: ' % char_space)

            # create a wide/narrow-seq (first digit=bars, second digit=spaces)
            seq = ''
            for s in range(0, len(bar_char[char_bar])):
                seq += bar_char[char_bar][s] + bar_char[char_space][s]

            for bar in range(0, len(seq)):
                # set line_width depending on value
                if seq[bar] == 'n':
                    line_width = narrow
                else:
                    line_width = wide

                # draw every second value, the other is represented by space
                if bar % 2 == 0:
                    self.rect(x, y, line_width, h, 'F')

                x += line_width
```

```python
     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': 'wwnnnnnnw', 'V': 'nwwnnnnnw', 'W': 'wwwnnnnnn',
             'X': 'nwnnwnnnw', 'Y': 'wwnnwnnnn', 'Z': 'nwwnwnnnn',
             '-': 'nwnnnnwnw', '.': 'wwnnnnwnn', ' ': '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']
```