```
#!/usr/bin/env python

""" Python Highlighter for PDF                    Version: 0.5

    py2pdf.py [options] <file1> [<file2> ...]

    options:
     -h  or  --help    print help (this message)
     -                 read from stdin (writes to stdout)
     --stdout          read from file, write to stdout
                         (restricted to first file only)
     --title=<title>   specify title
     --config=<file>   read configuration options from <file>
     --input=<type>    set input file type
                         'python': Python code (default)
                         'ascii':  arbitrary ASCII files (b/w)
     --mode=<mode>     set output mode
                         'color': output in color (default)
                         'mono':  output in b/w
     --paperFormat=    set paper format (ISO A, B, C series,
       <format>          US legal & letter; default: 'A4')
                         e.g. 'letter', 'A3', 'A4', 'B5', 'C6', ...
     --paperSize=      set paper size in points (size being a valid
       <size>          numeric 2-tuple (x,y) w/o any whitespace)
     --landscape       set landscape format (default: portrait)
     --bgCol=<col>     set page background-color in hex code like
                         '#FFA024' or '0xFFA024' for RGB components
                         (overwrites mono mode)
     --<cat>Col=<col>  set color of certain code categories, i.e.
                         <cat> can be the following:
                         'comm':  comments
                         'ident': identifiers
                         'kw':    keywords
                         'strng': strings
                         'param': parameters
                         'rest':  all the rest
     --fontName=<name> set base font name (default: 'Courier')
                         like 'Helvetica', 'Times-Roman'
     --fontSize=<size> set font size (default: 8)
     --tabSize=<size>  set tab size (default: 4)
     --lineNum         print line numbers
     --multiPage       generate one file per page (with filenames
                         tagged by 1, 2...), disables PDF outline
     --noOutline       don't generate PDF outline (default: unset)
     -v  or  --verbose set verbose mode

    Takes the input, assuming it is Python source code and formats
    it into PDF.

    * Uses Just van Rossum's PyFontify version 0.3.3 to tag Python
      scripts. You can get it via his homepage on the starship:
          http://starship.python.net/crew/just

    * Uses the ReportLab library version 0.92 (from 2000-04-10) to
      generate PDF. You can get it without charge from ReportLab:
          http://www.reportlab.com

    * Parts of this code still borrow heavily from Marc-Andre
      Lemburg's py2html who has kindly given permission to
      include them in py2pdf. Thanks, M.-A.!
"""

__copyright__ = """
---------------------------------------------------------------------
```

```
(c) Copyright by Dinu C. Gherman, 2000  (gherman@europemail.com)

    Permission to use, copy, modify, and distribute this
    software and its documentation for any purpose and
    without fee or royalty is hereby granted, provided
    that the above copyright notice appear in all copies
    and that both that copyright notice and this permission
    notice appear in supporting documentation or portions
    thereof, including modifications, that you make.

    THE AUTHOR DINU C. GHERMAN DISCLAIMS ALL WARRANTIES
    WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED
    WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT
    SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT
    OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER
    RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
    IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
    ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
    OR PERFORMANCE OF THIS SOFTWARE!
"""

__version__ = '0.5'
__author__  = 'Dinu C. Gherman'
__date__    = '2000-05-08'
__url__     = 'http://starship.python.net/crew/gherman/programs/py2pdf'


import sys, string, re, os, getopt

from reportlab.pdfgen import canvas
from reportlab.lib.colors import Color, HexColor
from reportlab.lib import fonts
from reportlab.lib.units import cm, inch
#from PyFontify import *


### Helpers functions.

def makeTuple(aString):
    """Evaluate a string securely into a tuple.

    Match the string and return an evaluated object thereof if and
    only if we think it is a tuple of two or more numeric decimal(!)
    values, integers or floats. E-notation, hex or octal is not
    supported, though! Shorthand notation (omitting leading or trail-
    zeros, before or after the decimal point) like .25 or 25. is
    supported.
    """

    c = string.count(aString, ',')
    num = '(\d*?\.?\d+?)|(\d+?\.?\d*?)'
    tup = string.join([num]*c, ',')

    if re.match('\(' + tup + '\)', aString):
        return eval(aString)
    else:
        details = '%s cannot be parsed into a numeric tuple!' % aString
        raise 'ValueError', details


def loadFontifier(options=None):
    "Load a tagging module and return a corresponding function."

    # We can't use 'if options' because of the modified
```

```
    # attribute lookup it seems.

    if type(options) != type(None) and options.marcs:
        # Use mxTextTool's tagging engine.

        from mxTextTools import tag
        from mxTextTools.Examples.Python import python_script
        tagFunc = lambda text, tag=tag, pytable=python_script: \
                  tag(text,pytable)[1]

    else:
        # Load Just's.

        try:
            import PyFontify

            if PyFontify.__version__ < '0.3':
                raise ValueError

            tagFunc = PyFontify.fontify

        except:
            print """
    Sorry, but this script needs the PyFontify.py module version 0.3;
    You can download it from Just's homepage at

       URL: http://starship.python.net/crew/just
"""
            sys.exit()

    return tagFunc


def makeColorFromString(aString):
    """Convert a string to a ReportLab RGB color object.

    Supported formats are: '0xFFFFFF', '#FFFFFF' and '(R,G,B)'
    where the latter is a tuple of three floats in the range
    [0.0, 1.0].
    """

    s = aString

    if s[0] == '#' or s[:2] in ('0x', '0X'):
        if s[:2] in ('0x', '0X'):
            return HexColor('#' + s[2:])

    elif s[0] == '(':
        r, g, b = makeTuple(aString)
        return Color(r, g, b)


### Utility classes.

# For py2pdf this is some kind of overkill, but it's fun, too.
class PaperFormat:
    """Class to capture a paper format/size and its orientation.

    This class represents an abstract paper format, including
    the size and orientation of a sheet of paper in some formats
    plus a few operations to change its size and orientation.
    """

    _A4W, _A4H = 21*cm, 29.7*cm
```

```
A0 = (4*_A4W, 4*_A4H)

_B4W, _B4H = 25*cm, 35.3*cm
B0 = (4*_B4W, 4*_B4H)

_C4W, _C4H = 22.9*cm, 32.4*cm
C0 = (4*_C4W, 4*_C4H)

letter = (8.5*inch, 11*inch)
legal = (8.5*inch, 14*inch)


def __init__(self, nameOrSize='A4', landscape=0):
    "Initialisation."

    t = type(nameOrSize)

    if t == type(''):
        self.setFormatName(nameOrSize, landscape)
    elif t == type((1,)):
        self.setSize(nameOrSize)
        self.setLandscape(landscape)


def __repr__(self):
    """Return a string representation of ourself.

    The returned string can also be used to recreate
    the same PaperFormat object again.
    """

    if self.name != 'custom':
        nos = `self.name`
    else:
        nos = `self.size`

    format = "PaperFormat(nameOrSize=%s, landscape=%d)"
    tuple = (nos, self.landscape)

    return format % tuple


def setSize(self, size=None):
    "Set explicit paper size."

    self.name = 'custom'
    x, y = self.size = size
    self.landscape = x < y


def setLandscape(self, flag):
    "Set paper orientation as desired."

    # Swap paper orientation if needed.

    self.landscape = flag
    x, y = self.size

    ls = self.landscape
    if (ls and x < y) or (not ls and x > y):
        self.size = y, x


def setFormatName(self, name='A4', landscape=0):
```

```
        "Set paper size derived from a format name."

        if name[0] in 'ABC':
            # Assume ISO-A, -B, -C series.
            # (Hmm, are B and C really ISO standards
            # or just DIN? Well...)
            c, f = name[0], int(name[1:])
            self.size = getattr(self, c + '0')
            self.name = c + '0'
            self.makeHalfSize(f)

        elif name == 'letter':
            self.size = self.letter
            self.name = 'letter'

        elif name == 'legal':
            self.size = self.legal
            self.name = 'legal'

        self.setLandscape(landscape)


    def makeHalfSize(self, times=1):
        "Reduce paper size (surface) by 50% multiple times."

        # Orientation remains unchanged.

        # Iterates only for times >= 1.
        for i in xrange(times):
            s = self.size
            self.size = s[1] / 2.0, s[0]

            if self.name[0] in 'ABC':
                self.name = self.name[0] + `int(self.name[1:]) + 1`
            else:
                self.name = 'custom'


    def makeDoubleSize(self, times=1):
        "Increase paper size (surface) by 50% multiple times."

        # Orientation remains unchanged.

        # Iterates only for times >= 1.
        for i in xrange(times):
            s = self.size
            self.size = s[1], s[0] * 2.0

            if self.name[0] in 'ABC':
                self.name = self.name[0] + `int(self.name[1:]) - 1`
            else:
                self.name = 'custom'


class Options:
    """Container class for options from command line and config files.

    This class is a container for options as they are specified
    when a program is called from a command line, but also from
    the same kind of options that are saved in configuration
    files. Both the short UNIX style (e.g. '-v') as well as the
    extended GNU style (e.g. '--help') are supported.

    An 'option' is a <name>/<value> pair with both parts being
```

```
    strings at the beginning, but where <value> might be conver-
    ted later into any other Python object.

    Option values can be accessed by using their name as an attri-
    bute of an Options object, returning None if no such option
    name exists (that makes None values impossible, but so what?).
    """

    # Hmm, could also use UserDict... maybe later.

    def __init__(self):
        "Initialize with some default options name/values."

        # Hmm, could also pass an initial optional dict...

        # Create a dictionary containing the options
        # and populate it with defaults.
        self.pool = {}
        self.setDefaults()


    def __getattr__(self, name):
        "Turn attribute access into dictionary lookup."

        return self.pool.get(name)


    def setDefaults(self):
        "Set default options."

        ### Maybe get these from a site config file...
        self.pool.update({'fontName' : 'Courier',
            'fontSize'   : 8,
            'bgCol'      : Color(1, 1, 1),
            'mode'       : 'color',
            'lineNum'    : 0,
            'tabSize'    : 4,
            'paperFormat': 'A4',
            'landscape'  : 0,
            'title'      : None,
            'multiPage'  : 0})

        # Default colors (for color mode), mostly taken from py2html.
        self.pool.update({'commCol' : HexColor('#1111CC'),
            'kwCol'    : HexColor('#3333CC'),
            'identCol' : HexColor('#CC0000'),
            'paramCol' : HexColor('#000066'),
            'strngCol' : HexColor('#119911'),
            'restCol'  : HexColor('#000000')})

        # Add a default 'real' paper format object.
        pf = PaperFormat(self.paperFormat, self.landscape)
        self.pool.update({'realPaperFormat' : pf})
        self.pool.update({'files' : []})


    def display(self):
        "Display all current option names and values."

        self.saveToFile(sys.stdout)


    def saveToFile(self, path):
        "Save options as a log file."
```

```
        if type(path) == type(''):
            f = open(path, 'w')
        else:
            # Assume a file-like object.
            f = path

        items = self.pool.items()
        items.sort()

        for n, v in items:
            f.write("%-15s : %s\n" % (n, `v`))


    def updateWithContentsOfFile(self, path):
        """Update options as specified in a config file.

        The file is expected to contain one option name/value pair
        (seperated by an equal sign) per line, but no other whitespace.
        Option values may contain equal signs, but no whitespace.
        Option names must be valid Python strings, preceeded by one
        or two dashes.
        """

        config = open(path).read()
        config = string.split(config, '\n')

        for cfg in config:
            if cfg == None:
                break

            if cfg == '' or cfg[0] == '#':
                continue

            # GNU long options
            if '=' in cfg and cfg[:2] == '--':
                p = string.find(cfg, '=')
                opt, arg = cfg[2:p], cfg[p+1:]
                self.updateOption(opt, arg)

            # Maybe treat single-letter options as well?
            # elif ':' in cfg and cfg[0] == '-' and cfg[1] != '-':
            #    pass

            else:
                self.updateOption(cfg[2:], None)


    def updateWithContentsOfArgv(self, argv):
        "Update options as specified in a (command line) argument vector."

        # Specify accepted short option names (UNIX style).
        shortOpts = 'hv'

        # Specify accepted long option names (GNU style).
        lo = 'tabSize= paperFormat= paperSize= landscape stdout title= fontName= fontSize='
        lo = lo + ' bgCol= lineNum marcs help multiPage noOutline config= input= mode='
        lo = lo + ' commCol= identCol= kwCol= strngCol= paramCol= restCol='
        longOpts = string.split(lo, ' ')

        try:
            optList, args = getopt.getopt(argv, shortOpts, longOpts)
        except getopt.error, msg:
            sys.stderr.write("%s\nuse -h or --help for help\n" % str(msg))
```

```
            sys.exit(2)

        self.updateOption('files', args) # Hmm, really needed?

        for o, v in optList:
            # Remove leading dashes (max. two).
            if o[0] == '-':
                o = o[1:]
            if o[0] == '-':
                o = o[1:]

            self.updateOption(o, v)


    def updateOption(self, name, value):
        "Update an option from a string value."

        # Special treatment for coloring options...
        if name[-3:] == 'Col':
            if name[:-3] in string.split('bg comm ident kw strng param rest', ' '):
                self.pool[name] = makeColorFromString(value)

        elif name == 'paperSize':
            tup = makeTuple(value)
            self.pool['paperSize'] = tup
            if not self.realPaperFormat:
                pf = PaperFormat(self.paperFormat, self.landscape)
                self.pool['realPaperFormat'] = pf
            self.pool['realPaperFormat'].setSize(tup)

        elif name == 'paperFormat':
            self.pool['paperFormat'] = value
            if not self.realPaperFormat:
                pf = PaperFormat(self.paperFormat, self.landscape)
                self.pool['realPaperFormat'] = pf
            self.pool['realPaperFormat'].setFormatName(self.paperFormat, self.landscape)

        elif name == 'landscape':
            self.pool['landscape'] = 1
            if self.realPaperFormat:
                self.pool['realPaperFormat'].setLandscape(1)

        elif name == 'fontSize':
            self.pool['fontSize'] = int(value)

        elif name == 'tabSize':
            self.pool['tabSize'] = int(value)

        elif name == 'mode':
            self.pool['mode'] = value
            if value == 'mono':
                cats = 'comm ident kw strng param rest'
                for cat in string.split(cats, ' '):
                    self.pool[cat + 'Col'] = Color(0, 0, 0)

        # Parse configuration file...
        elif name == 'config':
            self.updateWithContentsOfFile(value)

        elif name == 'stdout':
            self.pool['stdout'] = 1

        elif name == 'files':
            self.pool['files'] = value
```

```
        else:
            # Set the value found or 1 for options without values.
            self.pool[name] = value or 1


    def update(self, **options):
        "Update options."

        # Not much tested and/or used, yet!!

        for n, v in options.items():
            self.pool[n] = v


### Layouting classes.

class PDFLayouter:
    """A class to layout a simple PDF document.

    This is intended to help generate PDF documents where all pages
    follow the same kind of 'template' which is supposed to be the
    same adornments (header, footer, etc.) on each page plus a main
    'frame' on each page. These frames are 'connected' such that one
    can add individual text lines one by one with automatic line
    wrapping, page breaks and text flow between frames.
    """

    def __init__(self, options):
        "Initialisation."

        self.options = options
        self.canvas = None
        self.multiLineStringStarted = 0
        self.lineNum = 0

        # Set a default color and font.
        o = self.options
        self.currColor = o.restCol
        self.currFont = (o.fontName, o.fontSize)


    ### Helper methods.

    def setMainFrame(self, frame=None):
        "Define the main drawing frame of interest for each page."

        if frame:
            self.frame = frame
        else:
            # Maybe a long-term candidate for additional options...
            width, height = self.options.realPaperFormat.size
            self.frame = height - 3*cm, 3*cm, 2*cm, width - 2*cm

            # self.frame is a 4-tuple:
            # (topMargin, bottomMargin, leftMargin, rightMargin)


    def setPDFMetaInfo(self):
        "Set PDF meta information."

        o = self.options
        c = self.canvas
        c.setAuthor('py2pdf %s' % __version__)
```

```
        c.setSubject('')

        # Set filename.
        filename = ''

        # For stdin use title option or empty...
        if self.srcPath == sys.stdin:
            if o.title:
                filename = o.title
        # otherwise take the input file's name.
        else:
            path = os.path.basename(self.srcPath)
            filename = o.title or path

        c.setTitle(filename)


    def setFillColorAndFont(self, color, font):
        "Set new color/font (maintaining the current 'state')."

        self.currFont = font
        self.currColor = color

        fontName, fontSize = font
        self.text.setFont(fontName, fontSize)
        self.text.setFillColor(color)


    ### API

    def begin(self, srcPath, numLines):
        "Things to do before doing anything else."

        self.lineNum = 0
        self.pageNum = 0
        self.numLines = numLines
        self.srcPath = srcPath

        # Set output filename (stdout if desired).
        o = self.options
        if o.stdout:
            self.pdfPath = sys.stdout
        else:
            if srcPath != sys.stdin:
                self.pdfPath = os.path.splitext(srcPath)[0] + '.pdf'
            else:
                self.pdfPath = sys.stdout


    def beginDocument(self):
        """Things to do when a new document should be started.

        The initial page counter is 0, meaning that beginPage()
        will be called by beginLine()...
        """

        # Set initial page number and store file name.
        self.pageNum = 0
        o = self.options

        if not o.multiPage:
            # Create canvas.
            size = o.realPaperFormat.size
            self.canvas = canvas.Canvas(self.pdfPath, size, verbosity=0)
```

```
            c = self.canvas
            c.setPageCompression(1)
            c.setFont(o.fontName, o.fontSize)

            # Create document meta information.
            self.setPDFMetaInfo()

            # Set drawing frame.
            self.setMainFrame()

            # Determine the left text margin by adding the with
            # of the line number to the left margin of the main frame.
            format = "%%%dd " % len(`self.numLines`)
            fn, fs = self.currFont
            text = format % self.lineNum
            tm, bm, lm, rm = self.frame
            self.txm = lm
            if o.lineNum:
                self.txm = self.txm + c.stringWidth(text, fn, fs)


    def beginPage(self):
        "Things to do when a new page has to be added."

        o = self.options
        self.pageNum = self.pageNum + 1

        if not o.multiPage:
            tm, bm, lm, rm = self.frame
            self.text = self.canvas.beginText(lm, tm - o.fontSize)
            self.setFillColorAndFont(self.currColor, self.currFont)
        else:
            # Fail if stdout desired (with multiPage).
            if o.stdout:
                raise "IOError", "Can't create multiple pages on stdout!"

            # Create canvas with a modified path name.
            base, ext = os.path.splitext(self.pdfPath)
            newPath = "%s-%d%s" % (base, self.pageNum, ext)
            size = o.realPaperFormat.size
            self.canvas = canvas.Canvas(newPath, size, verbosity=0)
            c = self.canvas
            c.setPageCompression(1)
            c.setFont(o.fontName, o.fontSize)

            # Create document meta information.
            self.setPDFMetaInfo()

            # Set drawing frame.
            self.setMainFrame()

            tm, bm, lm, rm = self.frame
            self.text = self.canvas.beginText(lm, tm - o.fontSize)
            self.setFillColorAndFont(self.currColor, self.currFont)

        self.putPageDecoration()


    def beginLine(self, wrapped=0):
        "Things to do when a new line has to be added."

        # If there is no page yet, create the first one.
        if self.pageNum == 0:
            self.beginPage()
```

```
        # If bottom of current page reached, do a page break.
        # (This works only with one text object being used
        # for the entire page. Otherwise we need to maintain
        # the vertical position of the current line ourself.)
        y = self.text.getY()
        tm, bm, lm, rm = self.frame
        if y < bm:
            self.endPage()
            self.beginPage()

        # Print line number label, if needed.
        o = self.options
        if o.lineNum:
            #self.putLineNumLabel()
            font = ('Courier', o.fontSize)

            if not wrapped:
                # Print a label containing the line number.
                self.setFillColorAndFont(o.restCol, font)
                format = "%%%dd " % len(`self.numLines`)
                self.text.textOut(format % self.lineNum)
            else:
                # Print an empty label (using bgCol). Hackish!
                currCol = self.currColor
                currFont = self.currFont
                self.setFillColorAndFont(o.bgCol, font)
                self.text.textOut(' '*(len(`self.numLines`) + 1))
                self.setFillColorAndFont(currCol, currFont)


    def endLine(self, wrapped=0):
        "Things to do after a line is basically done."

        # End the current line by adding an 'end of line'.
        # (Actually done by the text object...)
        self.text.textLine('')

        if not wrapped:
            self.lineNum = self.lineNum + 1


    def endPage(self):
        "Things to do after a page is basically done."

        c = self.canvas

        # Draw the current text object (later we might want
        # to do that after each line...).
        c.drawText(self.text)
        c.showPage()

        if self.options.multiPage:
            c.save()


    def endDocument(self):
        "Things to do after the document is basically done."

        c = self.canvas

        # Display rest of last page and save it.
        c.drawText(self.text)
        c.showPage()
```

```
        c.save()


    def end(self):
        "Things to do after everything has been done."

        pass


    ### The real meat: methods writing something to a canvas.

    def putLineNumLabel(self, text, wrapped=0):
        "Add a long text that can't be split into chunks."

        o = self.options
        font = ('Courier', o.fontSize)

        if not wrapped:
            # Print a label containing the line number.
            self.setFillColorAndFont(o.restCol, font)
            format = "%%%dd " % len(`self.numLines`)
            self.text.textOut(format % self.lineNum)
        else:
            # Print an empty label (using bgCol). Hackish!
            currCol = self.currColor
            currFont = self.currFont
            self.setFillColorAndFont(o.bgCol, font)
            self.text.textOut(' '*(len(`self.numLines`) + 1))
            self.setFillColorAndFont(currCol, currFont)


    # Tried this recursively before, in order to determine
    # an appropriate string limit rapidly, but this wasted
    # much space and showed very poor results...
    # This linear method is very slow, but such lines should
    # be very rare, too!

    def putSplitLongText(self, text):
        "Add a long text that can't be split into chunks."

        # Now, the splitting will be with 'no mercy',
        # at the right margin of the main drawing area.

        M = len(text)
        t = self.text
        x = t.getX()
        o = self.options
        tm, bm, lm, rm = self.frame

        width = self.canvas.stringWidth
        fn, fs = self.currFont
        tw = width(text, fn, fs)

        tx = self.text
        if tw > rm - lm - x:
            i = 1
            T = ''
            while text:
                T = text[:i]
                tx = self.text # Can change after a page break.
                tw = width(T, fn, fs)

                if x + tw > rm:
                    tx.textOut(T[:-1])
```

```
                self.endLine(wrapped=1)
                self.beginLine(wrapped=1)
                x = tx.getX()
                text = text[i-1:]
                M = len(text)
                i = 0

            i = i + 1

            if i > M:
                break

        tx.textOut(T)

    else:
        t.textOut(text)


def putLongText(self, text):
    "Add a long text by gracefully splitting it into chunks."

    # Splitting is currently done only at blanks, but other
    # characters such as '.' or braces are also good
    # possibilities... later...

    o = self.options
    tm, bm, lm, rm = self.frame

    width = self.canvas.stringWidth
    fn, fs = self.currFont
    tw = width(text, fn, fs)

    arr = string.split(text, ' ')

    for i in range(len(arr)):
        a = arr[i]
        t = self.text # Can change during the loop...
        tw = width(a, fn, fs)
        x = t.getX()

        # If current item does not fit on current line, have it
        # split and then put before/after a line (maybe also
        # page) break.
        if x + tw > rm:
            self.putSplitLongText(a)
            t = self.text # Can change after a page break...

        # If it fits, just add it to the current text object.
        else:
            t.textOut(a)

        # Add the character we used to split th original text.
        if i < len(arr) - 1:
            t.textOut(' ')


def putText(self, text):
    "Add some text to the current line."

    t = self.text
    x = t.getX()
    o = self.options
    fn, fs = o.fontName, o.fontSize
    tw = self.canvas.stringWidth(text, fn, fs)
```

```
        rm = self.frame[3]

        if x + tw < rm:
            t.textOut(text)
        else:
            self.putLongText(text)


    # Not yet tested.
    def putLine(self, text):
        "Add a line to the current text."

        self.putText(text)
        self.endLine()


    def putPageDecoration(self):
        "Draw some decoration on each page."

        # Use some abbreviations.
        o = self.options
        c = self.canvas
        tm, bm, lm, rm = self.frame

        # Restore default font.
        c.setFont(o.fontName, o.fontSize)

        c.setLineWidth(0.5) # in pt.

        # Background color.
        c.setFillColor(o.bgCol)
        pf = o.realPaperFormat.size
        c.rect(0, 0, pf[0], pf[1], stroke=0, fill=1)

        # Header.
        c.setFillColorRGB(0, 0, 0)
        c.line(lm, tm + .5*cm, rm, tm + .5*cm)
        c.setFont('Times-Italic', 12)

        if self.pdfPath == sys.stdout:
            filename = o.title or ' '
        else:
            path = os.path.basename(self.srcPath)
            filename = o.title or path

        c.drawString(lm, tm + 0.75*cm + 2, filename)

        # Footer.
        c.line(lm, bm - .5*cm, rm, bm - .5*cm)
        c.drawCentredString(0.5 * pf[0], 0.5*bm, "Page %d" % self.pageNum)

        # Box around main frame.
        # c.rect(lm, bm, rm - lm, tm - bm)


class PythonPDFLayouter (PDFLayouter):
    """A class to layout a simple multi-page PDF document.
    """

    ### API for adding specific Python entities.

    def addKw(self, t):
        "Add a keyword."
```

```
        o = self.options

        # Make base font bold.
        fam, b, i = fonts.ps2tt(o.fontName)
        ps = fonts.tt2ps(fam, 1, i)
        font = (ps, o.fontSize)

        self.setFillColorAndFont(o.kwCol, font)

        # Do bookmarking...
        if not o.noOutline and not o.multiPage:
            if t in ('class', 'def'):
                tm, bm, lm, rm = self.frame
                pos = self.text.getX()

                if pos == self.txm:
                    self.startPositions = []

                self.startPos = pos

                if not hasattr(self, 'startPositions'):
                    self.startPositions = []

                if pos not in self.startPositions:
                    self.startPositions.append(pos)

                # Memorize certain keywords.
                self.itemFound = t

            else:
                self.itemFound = None
                self.startPos = None

        self.putText(t)


    def addIdent(self, t):
        "Add an identifier."

        o = self.options

        # Make base font bold.
        fam, b, i = fonts.ps2tt(o.fontName)
        ps = fonts.tt2ps(fam, 1, i)
        font = (ps, o.fontSize)

        self.setFillColorAndFont(o.identCol, font)
        self.putText(t)

        # Bookmark certain identifiers (class and function names).
        if not o.noOutline and not o.multiPage:
            item = self.itemFound
            if item:
                # Add line height to current vert. position.
                pos = self.text.getY() + o.fontSize

                nameTag = "p%sy%s" % (self.pageNum, pos)
                c = self.canvas
                i = self.startPositions.index(self.startPos)
                c.bookmarkHorizontalAbsolute0(nameTag, pos)
                c.addOutlineEntry0('%s %s' % (item, t), nameTag, i)


    def addParam(self, t):
```

```
        "Add a parameter."

        o = self.options
        font = (o.fontName, o.fontSize)
        self.setFillColorAndFont(o.paramCol, font)
        self.text.putText(t)


    def addSimpleString(self, t):
        "Add a simple string."

        o = self.options
        font = (o.fontName, o.fontSize)
        self.setFillColorAndFont(o.strngCol, font)
        self.putText(t)


    def addTripleStringBegin(self):
        "Memorize begin of a multi-line string."

        # Memorise that we started a multi-line string.
        self.multiLineStringStarted = 1


    def addTripleStringEnd(self, t):
        "Add a multi-line string."

        self.putText(t)

        # Forget about the multi-line string again.
        self.multiLineStringStarted = 0


    def addComm(self, t):
        "Add a comment."

        o = self.options

        # Make base font slanted.
        fam, b, i = fonts.ps2tt(o.fontName)
        ps = fonts.tt2ps(fam, b, 1)
        font = (ps, o.fontSize)

        self.setFillColorAndFont(o.commCol, font)
        self.putText(t)


    def addRest(self, line, eol):
        "Add a regular thing."

        o = self.options

        # Nothing else to be done, print line as-is...
        if line:
            font = (o.fontName, o.fontSize)
            self.setFillColorAndFont(o.restCol, font)

            # ... except if the multi-line-string flag is set, then we
            # decide to change the current color to that of strings and
            # just go on.
            if self.multiLineStringStarted:
                self.setFillColorAndFont(o.strngCol, font)

            self.putText(line)
```

```
        # Print an empty line.
        else:
            if eol != -1:
                self.putText('')

    ### End of API.


class EmptyPythonPDFLayouter (PythonPDFLayouter):
    """A PDF layout with no decoration and no margins.

    The main frame extends fully to all paper edges. This is
    useful for creating PDFs when writing one page per file,
    in order to provide pre-rendered, embellished Python
    source code to magazine publishers, who can include the
    individual files and only need to add their own captures.
    """

    def setMainFrame(self, frame=None):
        "Make a frame extending to all paper edges."

        width, height = self.options.realPaperFormat.size
        self.frame = height, 0, 0, width


    def putPageDecoration(self):
        "Draw no decoration at all."

        pass


### Pretty-printing classes.

class PDFPrinter:
    """Generic PDF Printer class.

    Does not do much, but write a PDF file created from
    any ASCII input file.
    """

    outFileExt = '.pdf'


    def __init__(self, options=None):
        "Initialisation."

        self.data = None     # Contains the input file.
        self.inPath = None   # Path of input file.
        self.Layouter = PDFLayouter

        if type(options) != type(None):
            self.options = options
        else:
            self.options = Options()


    ### I/O.

    def readFile(self, path):
        "Read the content of a file."

        if path == sys.stdin:
            f = path
```

```
        else:
            f = open(path)

        self.inPath = path

        data = f.read()
        o = self.options
        self.data = re.sub('\t', ' '*o.tabSize, data)
        f.close()


    def formatLine(self, line, eol=0):
        "Format one line of Python source code."

        font = ('Courier', 8)
        self.layouter.setFillColorAndFont(Color(0, 0, 0), font)
        self.layouter.putText(line)


    def writeData(self, srcCodeLines, inPath, outPath=None):
        "Convert Python source code lines into a PDF document."

        # Create a layouter object.
        self.layouter = self.Layouter(self.options)
        l = self.layouter

        # Loop over all tagged source lines, dissect them into
        # Python entities ourself and let the layouter do the
        # rendering.
        splitCodeLines = string.split(srcCodeLines, '\n')

        ### Must also handle the case of outPath being sys.stdout!!
        l.begin(inPath, len(splitCodeLines))
        l.beginDocument()

        for line in splitCodeLines:
            l.beginLine()
            self.formatLine(line)
            l.endLine()

        l.endDocument()
        l.end()


    def writeFile(self, data, inPath=None, outPath=None):
        "Write some data into a file."

        if inPath == sys.stdin:
            self.outPath = sys.stdout
        else:
            if not outPath:
                path = os.path.splitext(self.inPath)[0]
                self.outPath = path + self.outFileExt

        self.writeData(data, inPath, outPath or self.outPath)


    def process(self, inPath, outPath=None):
        "The real 'action point' for working with Pretty-Printers."

        self.readFile(inPath)
        self.writeFile(self.data, inPath, outPath)
```

```
class PythonPDFPrinter (PDFPrinter):
    """A class to nicely format tagged Python source code.

    """

    comm  = 'COMMENT'
    kw    = 'KEYWORD'
    strng = 'STRING'
    ident = 'IDENT'
    param = 'PARAMETER'

    outFileExt = '.pdf'


    def __init__(self, options=None):
        "Initialisation, calling self._didInit() at the end."

        if type(options) != type(None):
            self.options = options
        else:
            self.options = Options()

        self._didInit()


    def _didInit(self):
        "Post-Initialising"

        # Define regular expression patterns.
        s = self
        comp = re.compile

        s.commPat   = comp('(.*?)<'  + s.comm  + '>(.*?)</' + s.comm  + '>(.*)')
        s.kwPat     = comp('(.*?)<'  + s.kw    + '>(.*?)</' + s.kw    + '>(.*)')
        s.identPat  = comp('(.*?)<'  + s.ident + '>(.*?)</' + s.ident + '>(.*)')
        s.paramPat  = comp('(.*?)<'  + s.param + '>(.*?)</' + s.param + '>(.*)')
        s.stPat     = comp('(.*?)<'  + s.strng + '>(.*?)</' + s.strng + '>(.*)')
        s.strng1Pat = comp('(.*?)<'  + s.strng + '>(.*)')
        s.strng2Pat = comp('(.*?)</' + s.strng + '>(.*)')
        s.allPat    = comp('(.*)')

        cMatch  = s.commPat.match
        kMatch  = s.kwPat.match
        iMatch  = s.identPat.match
        pMatch  = s.paramPat.match
        sMatch  = s.stPat.match
        s1Match = s.strng1Pat.match
        s2Match = s.strng2Pat.match
        aMatch  = s.allPat.match

        self.matchList = ((cMatch, 'Comm'),
            (kMatch, 'Kw'),
            (iMatch, 'Ident'),
            (pMatch, 'Param'),
            (sMatch, 'SimpleString'),
            (s1Match, 'TripleStringBegin'),
            (s2Match, 'TripleStringEnd'),
            (aMatch, 'Rest'))

        self.Layouter = PythonPDFLayouter

        # Load fontifier.
        self.tagFunc = loadFontifier(self.options)
```

```
    ###

    def formatLine(self, line, eol=0):
        "Format one line of Python source code."

        # Values for eol: -1:no-eol, 0:dunno-yet, 1:do-eol.

        for match, meth in self.matchList:
            res = match(line)

            if res:
                groups = res.groups()
                method = getattr(self, '_format%s' % meth)
                method(groups, eol)
                break


    def _formatIdent(self, groups, eol):
        "Format a Python identifier."

        before, id, after = groups
        self.formatLine(before, -1)
        self.layouter.addIdent(id)
        self.formatLine(after, eol)


    def _formatParam(self, groups, eol):
        "Format a Python parameter."

        before, param, after = groups
        self.formatLine(before, -1)
        self.layouter.addParam(before)
        self.formatLine(after, eol)


    def _formatSimpleString(self, groups, eol):
        "Format a Python one-line string."

        before, s, after = groups
        self.formatLine(before, -1)
        self.layouter.addSimpleString(s)
        self.formatLine(after, eol)


    def _formatTripleStringBegin(self, groups, eol):
        "Format a Python multi-line line string (1)."

        before, after = groups
        self.formatLine(before, -1)
        self.layouter.addTripleStringBegin()
        self.formatLine(after, 1)


    def _formatTripleStringEnd(self, groups, eol):
        "Format a Python multi-line line string (2)."

        before, after = groups
        self.layouter.addTripleStringEnd(before)
        self.formatLine(after, 1)


    def _formatKw(self, groups, eol):
        "Format a Python keyword."
```

```
        before, kw, after = groups
        self.formatLine(before, -1)
        self.layouter.addKw(kw)
        self.formatLine(after, eol)


    def _formatComm(self, groups, eol):
        "Format a Python comment."

        before, comment, after = groups
        self.formatLine(before, -1)
        self.layouter.addComm(comment)
        self.formatLine(after, 1)


    def _formatRest(self, groups, eol):
        "Format a piece of a Python line w/o anything special."

        line = groups[0]
        self.layouter.addRest(line, eol)


    ###

    def writeData(self, srcCodeLines, inPath, outPath=None):
        "Convert Python source code lines into a PDF document."

        # Create a layouter object.
        self.layouter = self.Layouter(self.options)
        l = self.layouter

        # Loop over all tagged source lines, dissect them into
        # Python entities ourself and let the layouter do the
        # rendering.
        splitCodeLines = string.split(srcCodeLines, '\n')
        l.begin(inPath, len(splitCodeLines))
        l.beginDocument()

        for line in splitCodeLines:
            l.beginLine()
            self.formatLine(line)
            l.endLine()

        l.endDocument()
        l.end()


    def process(self, inPath, outPath=None):
        "The real 'action point' for working with Pretty-Printers."

        self.readFile(inPath)
        self.taggedData = self._fontify(self.data)
        self.writeFile(self.taggedData, inPath, outPath)


    ### Fontifying.

    def _fontify(self, pytext):
        ""

        formats = {
            'rest'       : ('', ''),
            'comment'    : ('<%s>' % self.comm,  '</%s>' % self.comm),
```

```
            'keyword'    : ('<%s>' % self.kw,    '</%s>' % self.kw),
            'parameter'  : ('<%s>' % self.param, '</%s>' % self.param),
            'identifier' : ('<%s>' % self.ident, '</%s>' % self.ident),
            'string'     : ('<%s>' % self.strng, '</%s>' % self.strng) }

        # Parse.
        taglist = self.tagFunc(pytext)

        # Prepend special 'rest' tag.
        taglist[:0] = [('rest', 0, len(pytext), None)]

        # Prepare splitting.
        splits = []
        self._addSplits(splits, pytext, formats, taglist)

        # Do splitting & inserting.
        splits.sort()
        l = []
        li = 0

        for ri, dummy, insert in splits:
            if ri > li:
                l.append(pytext[li:ri])

            l.append(insert)
            li = ri

        if li < len(pytext):
            l.append(pytext[li:])

        return string.join(l, '')


    def _addSplits(self, splits, text, formats, taglist):
        ""

        # Helper for fontify().
        for id, left, right, sublist in taglist:

            try:
                pre, post = formats[id]
            except KeyError:
                # msg = 'Warning: no format '
                # msg = msg + 'for %s specified\n'%repr(id)
                # sys.stderr.write(msg)
                pre, post = '', ''

            if type(pre) != type(''):
                pre = pre(text[left:right])

            if type(post) != type(''):
                post = post(text[left:right])

            # len(splits) is a dummy used to make sorting stable.
            splits.append((left, len(splits), pre))

            if sublist:
                self._addSplits(splits, text, formats, sublist)

            splits.append((right, len(splits), post))


### Main
```

```
def main(cmdline):
    "Process command line as if it were sys.argv"

    # Create default options and initialize with argv
    # from the command line.
    options = Options()
    options.updateWithContentsOfArgv(cmdline[1:])

    # Print help message if desired, then exit.
    if options.h or options.help:
        print __doc__
        sys.exit()

    # Apply modest consistency checks and exit if needed.
    cmdStr = string.join(cmdline, ' ')
    find = string.find
    if find(cmdStr, 'paperSize') >= 0 and find(cmdStr, 'paperFormat') >= 0:
        details = "You can specify either paperSize or paperFormat, "
        details = detail + "but not both!"
        raise 'ValueError', details

    # Create PDF converter and pass options to it.
    if options.input:
        input = string.lower(options.input)

        if input == 'python':
            P = PythonPDFPrinter
        elif input == 'ascii':
            P = PDFPrinter
        else:
            details = "Input file type must be 'python' or 'ascii'."
            raise 'ValueError', details

    else:
        P = PythonPDFPrinter

    p = P(options)

    # Display options if needed.
    if options.v or options.verbose:
        pass # p.options.display()

    # Start working.
    verbose = options.v or options.verbose

    if options.stdout:
        if len(options.files) > 1 and verbose:
            print "Warning: will only convert first file on command line."
        f = options.files[0]
        p.process(f, sys.stdout)
    else:
        if verbose:
            print 'py2pdf: working on:'

        for f in options.files:
            try:
                if verbose:
                    print '  %s' % f
                if f != '-':
                    p.process(f)
                else:
                    p.process(sys.stdin, sys.stdout)
            except IOError:
                if verbose:
```

```
                    print '(IOError!)',

        if verbose:
            print
            print 'Done.'


###

if __name__=='__main__':
    main(sys.argv)
```