Python In The REAL World Pt.107

PyFPDF is a library that allows you to generate PDF documents under Python. It was ported from FPDF ('Free'-PDF) to a PHP library. The repository is on github at https://github.com/reingart/pyfpdf and the documentation is at https://pyfpdf.readthedocs.io/en/latest/index.html

Some of the features that this library provides are the ability to include graphic images, positional printing, lines, rectangles, ellipses, headers and footers, and the ability to create templated forms for things like invoices.

It's rather old, and there hasn't been much activity recently. In fact, the last update to the repository was about 3 years ago.

You can easily install it with pip by using:

```
pip install fpdf
```

or you can clone the github repository and install it from there via pip like this:

```
pip install -e .
```

from the clone main folder.

Let's jump right in and create our first example program. Enter this program into your favorite IDE and save it as "ftest1.py". We'll make some changes to it later on.

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font('Arial', 'B', 16)
pdf.cell(40, 10, 'Hello Full Circle Magazine!')
pdf.output('tuto1a.pdf', 'F')
```

Now we'll break down the program. After we import the FPDF library, we instantiate the library, by calling pdf=FPDF(), with the default options.

Next, we add a page. These are pretty much the first two things that you have to do before you can do anything else.

We then set the default font for the page, then we use the cell method to print some simple text, and then we call the output method to create the PDF file itself.

When we created the PDF object, as I said, we used the default parameters. Here's the options:

```
pdf = FPDF(orientation, units, page format) where:

orientation = (p:portrait, l:landscape) (default = Portrait)

  units = (pt:point, mm:millimeter, cm:centimeter, in:inch) (default is mm)

 format = (A3, A4, A5, Letter, Legal) (default is A4) (see below)
```

If you need a custom page size, you can send a tuple with the width and height in the given units. If you are using Portrait mode, the order should be (width, height), but if you are using landscape, it should be (height, width). Also, try as I might, I couldn't get the units to work with the inch option. Nothing seems to render, so I stick with 'mm'.

Next we set the font to be used. The call is:

```
    fpdf.set_font(family, style='', size = 0)
```

The set_font method allows you to specify the font to be used for the next lines of text to be rendered. It's not quite as open as you might think though. There are five normal fonts you can use which are pre-defined (unless you use "add_font" first). These are:

- Courier (fixed-width)
- Helvetica or Arial
- Times
- Symbol
- ZapfDingbats

These 5 fonts provide fixed width, sans-serif, serif, and two symbolic fonts. The font family parameter is case insensitive as well as the font style. Those styles can be:

B : bold
I : italic
U : underline
empty : regular

Note: if you want to change the font size within the document,

without changing the family or style, you can call:

```
fpdf.set_font_size(size)
```

If you want to use a special font for some reason, outside of the standard font set provided, you can use the add_font method. However, it is fairly difficult, so we'll discuss it in a future article. In the meantime, you can look at the documentation to see how to do it.

Now, we'll take a look at the cell method, that allows you to place the text to be rendered. This method will print a rectangular area with optional borders, possibly background and foreground colors, and the character string. The text can be aligned to the right or left or can be centered. When the call is finished, the current position moves to the right or to the next line, depending on a parameter passed, and allows for an optional link to be attached to the text. The format of the method parameters is fairly comprehensive, but, luckily, the author of the library has set a number of defaults, so it isn't as bad as it could be.

```
fpdf.cell(w, h=0, txt='', border=0, ln=0, align='', fill=false, link='')
```

We'll take a look at each parameter in a bit more depth...

w: Cell width. If this value is 0, the cell will extend to the right margin.

h: The cell height. Default is 0.

txt: string to print

border: 0: no border, 1 : frame (or a string containing which lines of the frame to be rendered)

ln : 0: to the right, 1 to the beginning of the next line, 2: below

align: 'L' left align, 'C': Center, 'R' : Right align

fill: True: Background painted, False: Transparent. Default = False

link: URL or identifier returned by add_link()

Finally, we call pdf.output to render the file and save it as the filename specified (which is the 'F' parameter). There are other parameters that you can research in the document.

Once the program is done, you can open it with your default PDF viewer. It should look something like the image above.

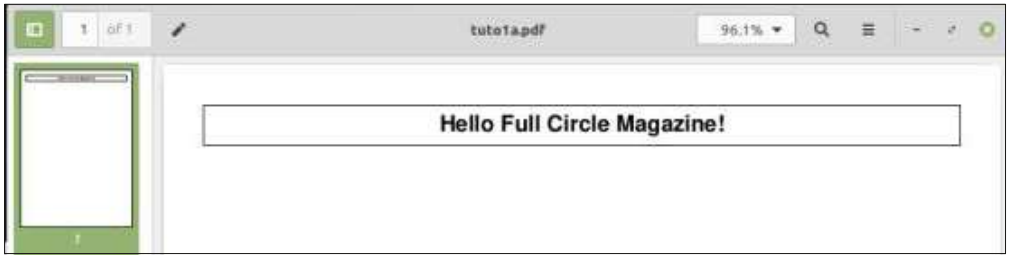
Now, we will take a look at adding the link parameter to the call method. Change the last two lines of your test program to:

```
pdf.cell(0, 10, 'Hello Full Circle Magazine!',
    1,1,'C',0,"http://fullcirclemagazine.org")
pdf.output('test1a.pdf', 'F')
```

Notice that we added all the parameters for the cell method. In this version, we will be adding a border with transparent fill, make the text centered, have the cell go from the far left extending to the far right of the page, set the next text line to be on the line below, and added a link to the Full Circle Magazine website, when you click on the text. We also changed the output file name.

Save the program as "test1a" and try it again (see image below).

That's great, but what if we want to do full paragraphs of text? There is a method that is close to the cell method that will handle this for us. It's called multi_cell. The multi_cell method uses the following parameters:



```
pdf.multi_cell(w: float, h:
float, txt: str, border = 0,

      align: str = 'J', fill:
bool = False)
```

It can be used in place of the cell method, but we are going to do something special to demonstrate this. This time, we will extend and override some of the builtin functions (that are simply stubs that are designed to be implemented in your code) which are header and footer, and add a couple of our own. Open another blank file in your IDE and name the file Demo3.py.

```
from fpdf import FPDF

import sys

import os
```

Of course, we have to start off with the import statements. Next, we'll extend the fpdf class by creating a header and footer method for ourselves (borrowed from one of the demos in the documentation). See top right.

The header method, as you might guess, creates a header that is (mostly) centered horizontally and consists of the title of our document. First, we set the font, then we use the get_string_width() method to calculate the width in whatever unit value that was set for the title when it would be rendered. That is then placed in a cell starting at the proper place at the top of the page. You could use a "0" in the cell method in place of the "w" and not use the set_x method, forcing the cell to start at a x position of 0 and extending to the right margin, but you really should see this alternate method.

```
class PDF(FPDF):
    def header(self):
        self.set_font('Arial', 'B', 15)
        # Calculate width of title and position
        w = self.get_string_width(title) + 6
        self.set_x((210 - w) / 2)
        # Title
        self.cell(w, 9, title, 0, 1, 'C', 0)
        # Line break
        self.ln(10)
```

The footer method handles placing the page number at the very bottom of the page, in this case (middle right) 15 mm from the bottom. You can set the color of the font, but I commented it out.

```
def footer(self):
  # Position at 1.5 cm from bottom
  self.set_y(-15)
  self.set_font('Arial', 'I', 8)
  # Text color in gray
  # self.set_text_color(128)
  # Page number
  self.cell(0, 10, 'Page ' + str(self.page_no()), 0, 0, 'C')
```

Next, we'll create a method to handle multi-line text. You can create different methods to handle different types of paragraphs. This, (below) again, was mostly borrowed from the documentation:

Now, we'll set some normal variables and add some properties to the PDF file. The properties are optional, but we'll do it here, just to show you how to do if you ever want to (these are outside of the class, so they are not indented):

Now, we send each group of text into the chapter_body() method we just created. For the

```
def chapter_body(self, name, fontfamily=None, fontattrib=None, fontsize=None):
    # Read text file
    with open(name, 'rb') as fh:
        txt = fh.read().decode('latin-1')
    if fontfamily == None:
        # Times 12
        self.set_font('Times', '', 12)
    else:
        self.set_font(fontfamily, fontattrib, fontsize)
    # Output justified text
    self.multi_cell(0, 5, txt)
    # Line break
    self.ln()
```

demo, I chose some text from last month's article and one of the programs to try to demonstrate how to use different font settings (top right).

```
pdf.chapter_body('demotext1.txt')

pdf.chapter_body('demotext2.txt','Arial','B',14)

pdf.chapter_body('birthdays2.py','Courier','B',11)
```

Now we render and save the PDF file:

```
pdf.output('demo3.pdf', 'F')
```

Finally, the following code (again borrowed from the documentation) will open the system default PDF viewer, assuming one is set, to display the PDF we just created. This saves you and the user the need to open a file manager window, dig around for the file and open it that way. I know the code works for Linux, but haven't tried it on a Windows machine or a Mac.

```
if sys.platform.startswith("linux"):

    os.system("xdg-open ./demo3.pdf")
```

```
title = 'Demo3 for Full Circle Magazine'
pdf = PDF('p', 'mm', 'letter')
pdf.alias_nb_pages()
pdf.set_title(title)
# Properties start here...
pdf.set_author('G.D. Walters')
pdf.set_subject('Demonstration program for Full Circle Magazine Issue #159')
pdf.set_keywords("PDF, Demonstration, Full Circle Magazine")
pdf.set_creator("Python")
# Properties end here...
pdf.add_page()
```

```
else:

    os.system("./demo3.pdf")
```

Now that the PDF is (hopefully) visible, check the properties of the document, after you check the text in the PDF itself. You should see something like this:

So there you have it. The beginnings of the ability to create your own PDF files. I strongly suggest that you download the repository with all the source code. It gives you a great insight into the abilities of the library. The biggest thing we didn't talk about this time is the ability of the library to use pre-defined templates. We'll save that for the next article.

The code files (and the text files for the last demo) have been uploaded to pastebin to make life easy for you. The links are below:

test1.py - https://pastebin.com/L2vUhAfa

test1a.py - https://pastebin.com/WsrGVPPU

demo3.py - https://pastebin.com/jaXSAJKg

demotext1.txt - https://pastebin.com/6FiWk7HF

demotext2.txt - https://pastebin.com/6AkCvMxx

birthdays2.py - https://pastebin.com/0Gwke6FD

Until next time; stay safe, healthy, positive and creative!

Greg Walters is a retired programmer living in Central Texas, USA. He has been a programmer since 1972 and in his spare time, he is an author, amateur photographer, luthier, fair musician and a pretty darn good cook. He still is the owner of RainyDaySolutions a consulting company and he spends most of his time writing articles for FCM and tutorials. His website is www.thedesignatedgeek.xyz .