fpdf2 manual

A minimalist PDF creation library for Python

v2.5.1

Table of contents

1. fpdf2

`fpdf2` is a library for PDF document generation in Python, forked from the unmaintained pyfpdf, itself ported from the PHP FPDF library.

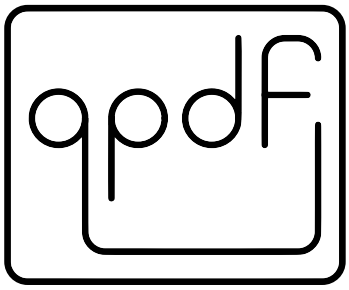
Latest Released Version: pypi v2.5.1

1.1 Main features

- Easy to use (and easy to extend)
- Small and compact code, useful for testing new features and teaching
- Many simple examples and scripts available in many languages
- PIL Integration for images (via Pillow)
- No installation, no compilation or other libraries (DLLs) required

This repository is a fork of the library's original port by Max Pat, with the following enhancements:

- Python 3.6+ support
- Unicode (UTF-8) TrueType font subset embedding (Central European, Cyrillic, Greek, Baltic, Thai, Chinese, Japanese, Korean, Hindi and almost any other language in the world)
- PNG, GIF and JPG support (including transparency and alpha channel)
- Shape, Line Drawing
- Arbitrary path drawing and basic SVG import
- Generate Code 39 & Interleaved 2 of 5 barcodes
- Cell / multi-cell / plaintext writing, automatic page breaks
- Basic conversion from HTML to PDF
- Images & links alternative descriptions
- Table of contents & document outline
- Optional basic Markdown-like styling: `**bold**, __italics__, --underlined--`
- Clean error handling through exceptions
- Only **one** dependency so far: Pillow
- Unit tests with `qpdf`-based PDF diffing
- PDF samples validation using 3 different checkers:



FPDF original features:

- Choice of measurement unit, page format and margins
- Page header and footer management
- Automatic page break, line break and text justification
- Image, colors and links support
- Page compression

1.2 Tutorials

- English
- Deutsch
- Italian
- español
- français
-
- português
- Русский

1.3 Installation

- From PyPI: `pip install fpdf2`
- From source:
- Clone the repository: `git clone https://github.com/PyFPDF/fpdf2.git`
- On ubuntu the following packages are required: `sudo apt-get install libjpeg-dev libpython-dev zlib1g-dev`
- Run `python setup.py install`

Developement: check the dedicated documentation page.

1.4 Community

1.4.1 Support

For community support, please feel free to file an issue or open a discussion.

1.4.2 They use fpdf2

- Undying Dusk : a **video game in PDF format**, with a gameplay based on exploration and logic puzzles, in the tradition of dungeon crawlers
- OpenDroneMap : a command line toolkit for processing aerial drone imagery
- OpenSfM : a Structure from Motion library, serving as a processing pipeline for reconstructing camera poses and 3D scenes from multiple images
- RPA Framework : libraries and tools for Robotic Process Automation (RPA), designed to be used with both Robot Framework
- Concordia : a platform developed by the US Library of Congress for crowdsourcing transcription and tagging of text in digitized images
- GovReady-Q Compliance Server : GRC platform for highly automated, user-friendly, self-service compliance assessments and documentation (DevSecOps)
- csv2pdf : convert CSV files to PDF files easily

1.4.3 Related

- Create PDFs with Python : a series of tutorial videos by bvalgard
- digidigital/Extensions-and-Scripts-for-pyFPDF-fpdf2 : scripts ported from PHP to add transpareny to elements of the page or part of an image, allow to write circular text, draw pie charts and bar diagrams, embed JavaScript, draw rectangles with rounded corners, draw a star shape, restrict the rendering of some elements to screen or printout, paint linear / radial / multi-color gradients gradients, add stamps & watermarks, write sheared text...

- Looking for alternative libraries? Check out this detailed list of PDF-related Python libs by Patrick Maupin (`pdfrw` author). There is also borb, pikepdf & WeasyPrint. We have some documentations about combining `fpdf2` with `borb` & `pdfrw`.

1.5 Misc

- Release notes: CHANGELOG.md
- This library could only exist thanks to the dedication of many volunteers around the world: list & map of contributors
- You can download an offline PDF version of this manual: fpdf2-manual.pdf

2. Tutorial

Version en français : Tutorial-fr

Deutsche Version: Tutorial-de

Versión en español: Tutorial-es

: Tutorial-

Versione in italiano: Tutorial-it

Versão em português: Tutorial-pt

Версия на русском: Tutorial-ru

Methods full documentation: `fpdf.FPDF` API doc

2.1 Tuto 1 - Minimal Example

Let's start with the classic example:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font("helvetica", "B", 16)
pdf.cell(40, 10, "Hello World!")
pdf.output("tuto1.pdf")
```

Resulting PDF

After including the library file, we create an `FPDF` object. The FPDF constructor is used here with the default values: pages are in A4 portrait and the measure unit is millimeter. It could have been specified explicitly with:

```
pdf = FPDF(orientation="P", unit="mm", format="A4")
```

It is possible to set the PDF in landscape mode (`L`) or to use other page formats (such as `Letter` and `Legal`) and measure units (`pt` , `cm` , `in`).

There is no page for the moment, so we have to add one with add_page. The origin is at the upper-left corner and the current position is by default placed at 1 cm from the borders; the margins can be changed with set_margins.

Before we can print text, it is mandatory to select a font with set_font, otherwise the document would be invalid. We choose Helvetica bold 16:

```
pdf.set_font('helvetica', 'B', 16)
```

We could have specified italics with `I` , underlined with `U` or a regular font with an empty string (or any combination). Note that the font size is given in points, not millimeters (or another user unit); it is the only exception. The other built-in fonts are `Times` , `Courier` , `Symbol` and `ZapfDingbats` .

We can now print a cell with cell. A cell is a rectangular area, possibly framed, which contains some text. It is rendered at the current position. We specify its dimensions, its text (centered or aligned), if borders should be drawn, and where the current position moves after it (to the right, below or to the beginning of the next line). To add a frame, we would do this:

```
pdf.cell(40, 10, 'Hello World!', 1)
```

To add a new cell next to it with centered text and go to the next line, we would do:

```
pdf.cell(60, 10, 'Powered by FPDF.', ln=1, align='C')
```

Remark: the line break can also be done with ln. This method allows to specify in addition the height of the break.

Finally, the document is closed and saved under the provided file path using output. Without any parameter provided, `output()` returns the PDF `bytearray` buffer.

2.2 Tuto 2 - Header, footer, page break and image

Here is a two page example with header, footer and logo:

```
from fpdf import FPDF


class PDF(FPDF):
    def header(self):
        # Rendering logo:
        self.image("../docs/fpdf2-logo.png", 10, 8, 33)
        # Setting font: helvetica bold 15
        self.set_font("helvetica", "B", 15)
        # Moving cursor to the right:
        self.cell(80)
        # Printing title:
        self.cell(30, 10, "Title", 1, 0, "C")
        # Performing a line break:
        self.ln(20)

    def footer(self):
        # Position cursor at 1.5 cm from bottom:
        self.set_y(-15)
        # Setting font: helvetica italic 8
        self.set_font("helvetica", "I", 8)
        # Printing page number:
        self.cell(0, 10, f"Page {self.page_no()}/{{nb}}", 0, 0, "C")


# Instantiation of inherited class
pdf = PDF()
pdf.alias_nb_pages()
pdf.add_page()
pdf.set_font("Times", size=12)
for i in range(1, 41):
    pdf.cell(0, 10, f"Printing line number {i}", 0, 1)
pdf.output("tuto2.pdf")
```

Resulting PDF

This example makes use of the header and footer methods to process page headers and footers. They are called automatically. They already exist in the FPDF class but do nothing, therefore we have to extend the class and override them.

The logo is printed with the image method by specifying its upper-left corner and its width. The height is calculated automatically to respect the image proportions.

To print the page number, a null value is passed as the cell width. It means that the cell should extend up to the right margin of the page; it is handy to center text. The current page number is returned by the page_no method; as for the total number of pages, it is obtained by means of the special value `{nb}` which will be substituted on document closure (provided you first called alias_nb_pages). Note the use of the set_y method which allows to set position at an absolute location in the page, starting from the top or the bottom.

Another interesting feature is used here: the automatic page breaking. As soon as a cell would cross a limit in the page (at 2 centimeters from the bottom by default), a break is performed and the font restored. Although the header and footer select their own font (`helvetica`), the body continues with `Times`. This mechanism of automatic restoration also applies to colors and line width. The limit which triggers page breaks can be set with set_auto_page_break.

2.3 Tuto 3 - Line breaks and colors

Let's continue with an example which prints justified paragraphs. It also illustrates the use of colors.

```
from fpdf import FPDF


class PDF(FPDF):
    def header(self):
        # Setting font: helvetica bold 15
        self.set_font("helvetica", "B", 15)
        # Calculating width of title and setting cursor position:
        width = self.get_string_width(self.title) + 6
        self.set_x((210 - width) / 2)
        # Setting colors for frame, background and text:
        self.set_draw_color(0, 80, 180)
        self.set_fill_color(230, 230, 0)
        self.set_text_color(220, 50, 50)
        # Setting thickness of the frame (1 mm)
        self.set_line_width(1)
        # Printing title:
        self.cell(width, 9, self.title, 1, 1, "C", True)
        # Performing a line break:
        self.ln(10)

    def footer(self):
        # Setting position at 1.5 cm from bottom:
        self.set_y(-15)
        # Setting font: helvetica italic 8
        self.set_font("helvetica", "I", 8)
        # Setting text color to gray:
        self.set_text_color(128)
        # Printing page number
        self.cell(0, 10, f"Page {self.page_no()}", 0, 0, "C")

    def chapter_title(self, num, label):
        # Setting font: helvetica 12
        self.set_font("helvetica", "", 12)
        # Setting background color
        self.set_fill_color(200, 220, 255)
        # Printing chapter name:
        self.cell(0, 6, f"Chapter {num} : {label}", 0, 1, "L", True)
        # Performing a line break:
        self.ln(4)

    def chapter_body(self, filepath):
        # Reading text file:
        with open(filepath, "rb") as fh:
            txt = fh.read().decode("latin-1")
        # Setting font: Times 12
        self.set_font("Times", size=12)
        # Printing justified text:
        self.multi_cell(0, 5, txt)
        # Performing a line break:
        self.ln()
        # Final mention in italics:
        self.set_font(style="I")
        self.cell(0, 5, "(end of excerpt)")

    def print_chapter(self, num, title, filepath):
        self.add_page()
        self.chapter_title(num, title)
        self.chapter_body(filepath)


pdf = PDF()
pdf.set_title("20000 Leagues Under the Seas")
pdf.set_author("Jules Verne")
pdf.print_chapter(1, "A RUNAWAY REEF", "20k_c1.txt")
pdf.print_chapter(2, "THE PROS AND CONS", "20k_c1.txt")
pdf.output("tuto3.pdf")
```

Resulting PDF

Jules Verne text

The get_string_width method allows determining the length of a string in the current font, which is used here to calculate the position and the width of the frame surrounding the title. Then colors are set (via set_draw_color, set_fill_color and set_text_color) and the thickness of the line is set to 1 mm (against 0.2 by default) with set_line_width. Finally, we output the cell (the last parameter to true indicates that the background must be filled).

The method used to print the paragraphs is multi_cell. Text is justified by default. Each time a line reaches the right extremity of the cell or a carriage return character (`\n`) is met, a line break is issued and a new cell automatically created under the current

one. An automatic break is performed at the location of the nearest space or soft-hyphen (`\u00ad`) character before the right limit. A soft-hyphen will be replaced by a normal hyphen when triggering a line break, and ignored otherwise.

Two document properties are defined: the title (set_title) and the author (set_author). Properties can be viewed by two means. First is to open the document directly with Acrobat Reader, go to the File menu and choose the Document Properties option. The second, also available from the plug-in, is to right-click and select Document Properties.

2.4 Tuto 4 - Multi Columns

This example is a variant of the previous one, showing how to lay the text across multiple columns.

```
from fpdf import FPDF


class PDF(FPDF):
    def __init__(self):
        super().__init__()
        self.col = 0  # Current column
        self.y0 = 0  # Ordinate of column start

    def header(self):
        self.set_font("helvetica", "B", 15)
        width = self.get_string_width(self.title) + 6
        self.set_x((210 - width) / 2)
        self.set_draw_color(0, 80, 180)
        self.set_fill_color(230, 230, 0)
        self.set_text_color(220, 50, 50)
        self.set_line_width(1)
        self.cell(width, 9, self.title, 1, 1, "C", True)
        self.ln(10)
        # Saving ordinate position:
        self.y0 = self.get_y()

    def footer(self):
        self.set_y(-15)
        self.set_font("helvetica", "I", 8)
        self.set_text_color(128)
        self.cell(0, 10, f"Page {self.page_no()}", 0, 0, "C")

    def set_col(self, col):
        # Set column position:
        self.col = col
        x = 10 + col * 65
        self.set_left_margin(x)
        self.set_x(x)

    @property
    def accept_page_break(self):
        if self.col < 2:
            # Go to next column:
            self.set_col(self.col + 1)
            # Set ordinate to top:
            self.set_y(self.y0)
            # Stay on the same page:
            return False
        # Go back to first column:
        self.set_col(0)
        # Trigger a page break:
        return True

    def chapter_title(self, num, label):
        self.set_font("helvetica", "", 12)
        self.set_fill_color(200, 220, 255)
        self.cell(0, 6, f"Chapter {num} : {label}", 0, 1, "L", True)
        self.ln(4)
        # Saving ordinate position:
        self.y0 = self.get_y()

    def chapter_body(self, name):
        # Reading text file:
        with open(name, "rb") as fh:
            txt = fh.read().decode("latin-1")
        # Setting font: Times 12
        self.set_font("Times", size=12)
        # Printing text in a 6cm width column:
        self.multi_cell(60, 5, txt)
        self.ln()
        # Final mention in italics:
        self.set_font(style="I")
        self.cell(0, 5, "(end of excerpt)")
        # Start back at first column:
        self.set_col(0)

    def print_chapter(self, num, title, name):
        self.add_page()
        self.chapter_title(num, title)
        self.chapter_body(name)
```

```
pdf = PDF()
pdf.set_title("20000 Leagues Under the Seas")
pdf.set_author("Jules Verne")
pdf.print_chapter(1, "A RUNAWAY REEF", "20k_c1.txt")
pdf.print_chapter(2, "THE PROS AND CONS", "20k_c1.txt")
pdf.output("tuto4.pdf")
```

Resulting PDF

Jules Verne text

The key difference from the previous tutorial is the use of the accept_page_break and the set_col methods.

Using the accept_page_break method, once the cell crosses the bottom limit of the page, it will check the current column number. If it is less than 2 (we chose to divide the page in three columns) it will call the set_col method, increasing the column number and altering the position of the next column so the text may continue there.

Once the bottom limit of the third column is reached, the accept_page_break method will reset and go back to the first column and trigger a page break.

2.5 Tuto 5 - Creating Tables

This tutorial will explain how to create tables easily.

The code will create three different tables to explain what can be achieved with some simple adjustments.

```
import csv
from fpdf import FPDF


class PDF(FPDF):
    def basic_table(self, headings, rows):
        for heading in headings:
            self.cell(40, 7, heading, 1)
        self.ln()
        for row in rows:
            for col in row:
                self.cell(40, 6, col, 1)
            self.ln()

    def improved_table(self, headings, rows, col_widths=(42, 39, 35, 40)):
        for col_width, heading in zip(col_widths, headings):
            self.cell(col_width, 7, heading, 1, 0, "C")
        self.ln()
        for row in rows:
            self.cell(col_widths[0], 6, row[0], "LR")
            self.cell(col_widths[1], 6, row[1], "LR")
            self.cell(col_widths[2], 6, row[2], "LR", 0, "R")
            self.cell(col_widths[3], 6, row[3], "LR", 0, "R")
            self.ln()
        # Closure line:
        self.cell(sum(col_widths), 0, "", "T")

    def colored_table(self, headings, rows, col_widths=(42, 39, 35, 42)):
        # Colors, line width and bold font:
        self.set_fill_color(255, 100, 0)
        self.set_text_color(255)
        self.set_draw_color(255, 0, 0)
        self.set_line_width(0.3)
        self.set_font(style="B")
        for col_width, heading in zip(col_widths, headings):
            self.cell(col_width, 7, heading, 1, 0, "C", True)
        self.ln()
        # Color and font restoration:
        self.set_fill_color(224, 235, 255)
        self.set_text_color(0)
        self.set_font()
        fill = False
        for row in rows:
            self.cell(col_widths[0], 6, row[0], "LR", 0, "L", fill)
            self.cell(col_widths[1], 6, row[1], "LR", 0, "L", fill)
            self.cell(col_widths[2], 6, row[2], "LR", 0, "R", fill)
            self.cell(col_widths[3], 6, row[3], "LR", 0, "R", fill)
            self.ln()
            fill = not fill
        self.cell(sum(col_widths), 0, "", "T")


def load_data_from_csv(csv_filepath):
    headings, rows = [], []
    with open(csv_filepath, encoding="utf8") as csv_file:
```

```
        for row in csv.reader(csv_file, delimiter=","):
            if not headings:  # extracting column names from first row:
                headings = row
            else:
                rows.append(row)
    return headings, rows


col_names, data = load_data_from_csv("countries.txt")
pdf = PDF()
pdf.set_font("helvetica", size=14)
pdf.add_page()
pdf.basic_table(col_names, data)
pdf.add_page()
pdf.improved_table(col_names, data)
pdf.add_page()
pdf.colored_table(col_names, data)
pdf.output("tuto5.pdf")
```

Resulting PDF - Countries text

Since a table is just a collection of cells, it is natural to build one from them.

The first example is achieved in the most basic way possible: simple framed cells, all of the same size and left aligned. The result is rudimentary but very quick to obtain.

The second table brings some improvements: each column has its own width, titles are centered and figures right aligned. Moreover, horizontal lines have been removed. This is done by means of the border parameter of the Cell() method, which specifies which sides of the cell must be drawn. Here we want the left (L) and right (R) ones. Now only the problem of the horizontal line to finish the table remains. There are two possibilities to solv it: check for the last line in the loop, in which case we use LRB for the border parameter; or, as done here, add the line once the loop is over.

The third table is similar to the second one but uses colors. Fill, text and line colors are simply specified. Alternate coloring for rows is obtained by using alternatively transparent and filled cells.

2.6 Tuto 6 - Creating links and mixing text styles

This tutorial will explain several ways to insert links inside a pdf document, as well as adding links to external sources.

It will also show several ways we can use different text styles, (bold, italic, underline) within the same text.

```
from fpdf import FPDF, HTMLMixin


class MyFPDF(FPDF, HTMLMixin):
    pass


pdf = MyFPDF()

# First page:
pdf.add_page()
pdf.set_font("helvetica", size=20)
pdf.write(5, "To find out what's new in self tutorial, click ")
pdf.set_font(style="U")
link = pdf.add_link()
pdf.write(5, "here", link)
pdf.set_font()

# Second page:
pdf.add_page()
pdf.set_link(link)
pdf.image(
    "../docs/fpdf2-logo.png", 10, 10, 50, 0, "", "https://pyfpdf.github.io/fpdf2/"
)
pdf.set_left_margin(60)
pdf.set_font_size(18)
pdf.write_html(
    """You can print text mixing different styles using HTML tags: <b>bold</b>, <i>italic</i>,
<u>underlined</u>, or <b><i><u>all at once</u></i></b>!
<br><br>You can also insert links on text, such as <a href="https://pyfpdf.github.io/fpdf2/">https://pyfpdf.github.io/fpdf2/</a>,
or on an image: the logo is clickable!"""
)
pdf.output("tuto6.pdf")
```

Resulting PDF - fpdf2-logo

The new method shown here to print text is write() . It is very similar to multi_cell() , the key differences being:

- The end of line is at the right margin and the next line begins at the left margin.
- The current position moves to the end of the text.

The method therefore allows us to write a chunk of text, alter the font style, and continue from the exact place we left off. On the other hand, its main drawback is that we cannot justify the text like we do with the multi_cell() method.

In the first page of the example, we used write() for this purpose. The beginning of the sentence is written in regular style text, then using the set_font() method, we switched to underline and finished the sentence.

To add an internal link pointing to the second page, we used the add_link() method, which creates a clickable area which we named "link" that directs to another place within the document. On the second page, we used set_link() to define the destination area for the link we just created.

To create the external link using an image, we used image() . The method has the option to pass a link as one of its arguments. The link can be both internal or external.

As an alternative, another option to change the font style and add links is to use the `write_html()` method. It is an html parser, which allows adding text, changing font style and adding links using html.

3. Existing PDFs

`fpdf2` cannot **parse** existing PDF files.

However, other Python libraries can be combined with `fpdf2` in order to add new content to existing PDF files.

This page provides several examples of doing so using `pdfrw`, a great zero-dependency pure Python library dedicated to reading & writing PDFs, with numerous examples and a very clean set of classes modelling the PDF internal syntax.

3.1 Adding content onto an existing PDF page

```
import sys
from fpdf import FPDF
from pdfrw import PageMerge, PdfReader, PdfWriter

IN_FILEPATH = sys.argv[1]
OUT_FILEPATH = sys.argv[2]
ON_PAGE_INDEX = 1
UNDERNEATH = False  # if True, new content will be placed underneath page (painted first)

def new_content():
    fpdf = FPDF()
    fpdf.add_page()
    fpdf.set_font("helvetica", size=36)
    fpdf.text(50, 50, "Hello!")
    reader = PdfReader(fdata=bytes(fpdf.output()))
    return reader.pages[0]

writer = PdfWriter(trailer=PdfReader(IN_FILEPATH))
PageMerge(writer.pagearray[ON_PAGE_INDEX]).add(new_content(), prepend=UNDERNEATH).render()
writer.write(OUT_FILEPATH)
```

3.2 Adding a page to an existing PDF

```
import sys
from fpdf import FPDF
from pdfrw import PdfReader, PdfWriter

IN_FILEPATH = sys.argv[1]
OUT_FILEPATH = sys.argv[2]
NEW_PAGE_INDEX = 1  # set to None to append at the end

def new_page():
    fpdf = FPDF()
    fpdf.add_page()
    fpdf.set_font("helvetica", size=36)
    fpdf.text(50, 50, "Hello!")
    reader = PdfReader(fdata=bytes(fpdf.output()))
    return reader.pages[0]

writer = PdfWriter(trailer=PdfReader(IN_FILEPATH))
writer.addpage(new_page(), at_index=NEW_PAGE_INDEX)
writer.write(OUT_FILEPATH)
```

This example relies on pdfrw *Pull Request* #216. Until it is merged, you can install a forked version of `pdfrw` including the required patch:

```
pip install git+https://github.com/PyPDF/pdfrw.git@addpage_at_index
```

4. HTML

`fpdf2` supports basic rendering from HTML.

This is implemented by using `html.parser.HTMLParser` from the Python standard library. The whole HTML 5 specification is very likely **not** supported, neither as CSS, but bug reports & contributions are very welcome to improve this.

For a more robust & feature-full HTML-to-PDF converter in Python, you may want to check WeasyPrint.

4.1 write_html usage example

HTML rendering require the use of `fpdf.HTMLMixin`, that provides a new `write_html` method:

```
from fpdf import FPDF, HTMLMixin

class PDF(FPDF, HTMLMixin):
    pass

pdf = PDF()
pdf.add_page()
pdf.write_html("""
  <h1>Big title</h1>
  <section>
    <h2>Section title</h2>
    <p><b>Hello</b> world. <u>I am</u> <i>tired</i>.</p>
    <p><a href="https://github.com/PyFPDF/fpdf2">PyFPDF/fpdf2 GitHub repo</a></p>
    <p align="right">right aligned text</p>
    <p>i am a paragraph <br />in two parts.</p>
    <font color="#00ff00"><p>hello in green</p></font>
    <font size="7"><p>hello small</p></font>
    <font face="helvetica"><p>hello helvetica</p></font>
    <font face="times"><p>hello times</p></font>
  </section>
  <section>
    <h2>Other section title</h2>
    <ul><li>unordered</li><li>list</li><li>items</li></ul>
    <ol><li>ordered</li><li>list</li><li>items</li></ol>
    <br>
    <br>
    <pre>i am preformatted text.</pre>
    <br>
    <blockquote>hello blockquote</blockquote>
    <table width="50%">
      <thead>
        <tr>
          <th width="30%">ID</th>
          <th width="70%">Name</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>1</td>
          <td>Alice</td>
        </tr>
        <tr>
          <td>2</td>
          <td>Bob</td>
        </tr>
      </tbody>
    </table>
  </section>
""")
pdf.output("html.pdf")
```

4.2 Supported HTML features

- `<h1>` to `<h8>`: headings (and `align` attribute)
- `<p>`: paragraphs (and `align` attribute)
- `<b>`, `<i>`, `<u>`: bold, italic, underline
- `<font>`: (and `face`, `size`, `color` attributes)
- `<center>` for aligning
- `<a>`: links (and `href` attribute)

- `<img>`: images (and `src`, `width`, `height` attributes)
- `<ol>`, `<ul>`, `<li>`: ordered, unordered and list items (can be nested)
- `<table>`: (and `border`, `width` attributes)
- `<thead>`: header (opens each page)
- `<tfoot>`: footer (closes each page)
- `<tbody>`: actual rows
- `<tr>`: rows (with `bgcolor` attribute)
- `<th>`: heading cells (with `align`, `bgcolor`, `width` attributes)
- `<td>`: cells (with `align`, `bgcolor`, `width` attributes)

Notes:

- tables should have at least a first `<th>` row with a `width` attribute.
- currently multi-line text in table cells is not supported, *cf.* issue 91. Contributions are welcome to add support for this feature! 😊

5. Images

When rendering an image, its size on the page can be specified in several ways:

- explicit width and height (expressed in user units)
- one explicit dimension, the other being calculated automatically in order to keep the original proportions
- no explicit dimension, in which case the image is put at 72 dpi

Note that if an image is displayed several times, only one copy is embedded in the file.

5.1 Simple example

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.image("docs/fpdf2-logo.png", x=20, y=60)
pdf.output("pdf-with-image.pdf")
```

5.2 Assembling images

`fpdf2` can be an easy solution to assemble images into a PDF.

The following code snippets provide examples of some basic layouts for assembling images into PDF files.

5.2.1 Side by side images, full height, landscape page

```
from fpdf import FPDF

pdf = FPDF(orientation="landscape")
pdf.set_margin(0)
pdf.add_page()
pdf.image("imgA.png", h=pdf.eph, w=pdf.epw/2)               # full page height, half page width
pdf.set_y(0)
pdf.image("imgB.jpg", h=pdf.eph, w=pdf.epw/2, x=pdf.epw/2)  # full page height, half page width, right half of the page
pdf.output("side-by-side.pdf")
```

5.3 Alternative description

A textual description of the image can be provided, for accessibility purposes:

```
pdf.image("docs/fpdf2-logo.png", x=20, y=60, alt_text="Snake logo of the fpdf2 library")
```

5.4 Usage with Pillow

You can perform image manipulations using the Pillow library, and easily embed the result:

```
from fpdf import FPDF
from PIL import Image

pdf = FPDF()
pdf.add_page()
img = Image.open("docs/fpdf2-logo.png")
img = img.crop((10, 10, 490, 490)).resize((96, 96), resample=Image.NEAREST)
pdf.image(img, x=80, y=100)
pdf.output("pdf-with-image.pdf")
```

5.5 SVG images

SVG images passed to the `image()` method will be embedded as PDF paths:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.image("SVG_logo.svg", w=100)
pdf.output("pdf-with-vector-image.pdf")
```

5.6 Retrieve images from URLs

URLs to images can be directly passed to the `image()` method:

```
pdf.image("https://upload.wikimedia.org/wikipedia/commons/7/70/Example.png")
```

5.7 Image compression

By default, `fpdf2` will avoid altering your images : no image conversion from / to PNG / JPEG is performed.

However, you can easily tell `fpdf2` to convert and embed all images as JPEGs in order to reduce your PDF size:

```
from fpdf import FPDF

pdf = FPDF()
pdf.set_image_filter("DCTDecode")
pdf.add_page()
pdf.image("docs/fpdf2-logo.png", x=20, y=60)
pdf.output("pdf-with-image.pdf")
```

Beware that "flattening" images this way will convert alpha channels to black.

5.8 Oversized images detection & downscaling

If the resulting PDF size is a concern, you may want to check if some inserted images are *oversized*, meaning their resolution is unnecessarily high given the size they are displayed.

There is how to enable this detection mechanism with `fpdf2` :

```
pdf.oversized_images = "WARN"
```

After setting this property, a `WARNING` log will be displayed whenever an oversized image is inserted.

`fpdf2` is also able to automatically downscale such oversized images:

```
pdf.oversized_images = "DOWNSCALE"
```

After this, oversized images will be automatically resized, generating `DEBUG` logs like this:

```
OVERSIZED: Generated new low-res image with name=lowres-test.png dims=(319, 451) id=2
```

For finer control, you can set `pdf.oversized_images_ratio` to set the threshold determining if an image is oversized.

If the concepts of "image compression" or "image resolution" are a bit obscure for you, this article is a recommended reading: The 5 minute guide to image quality

5.9 Disabling transparency

By default images transparency is preserved: alpha channels are extracted and converted to an embedded `SMask`. This can be disabled by setting `.allow_images_transparency`, *e.g.* to allow compliance with PDF/A-1:

```python
from fpdf import FPDF

pdf = FPDF()
pdf.allow_images_transparency = False
pdf.set_font("Helvetica", size=15)
pdf.cell(w=pdf.epw, h=30, txt="Text behind. " * 6)
pdf.image("docs/fpdf2-logo.png", x=0)
pdf.output("pdf-including-image-without-transparency.pdf")
```

6. Links

`fpdf2` can generate both **internal** links (to other pages in the document) & **hyperlinks** (links to external URLs that will be opened in a browser).

6.1 Hyperlink with FPDF.cell

This method makes the whole cell clickable (not only the text):

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font("helvetica", size=24)
pdf.cell(w=40, h=10, txt="Cell link", border=1, align="C", link="https://github.com/PyFPDF/fpdf2")
pdf.output("hyperlink.pdf")
```

6.2 Hyperlink with FPDF.link

The `FPDF.link` is a low-level method that defines a rectangular clickable area.

There is an example showing how to place such rectangular link over some text:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font("helvetica", size=36)
line_height = 10
text = "Text link"
pdf.text(x=0, y=line_height, txt=text)
width = pdf.get_string_width(text)
pdf.link(x=0, y=0, w=width, h=line_height, link="https://github.com/PyFPDF/fpdf2")
pdf.output("hyperlink.pdf")
```

6.3 Hyperlink with write_html

An alternative method using `fpdf.HTMLMixin`:

```
from fpdf import FPDF, HTMLMixin

class PDF(FPDF, HTMLMixin):
    pass

pdf = PDF()
pdf.set_font_size(16)
pdf.add_page()
pdf.write_html('<a href="https://github.com/PyFPDF/fpdf2">Link defined as HTML</a>')
pdf.output("hyperlink.pdf")
```

The hyperlinks defined this way will be rendered in blue with underline.

6.4 Internal links

Using `FPDF.cell`:

```
from fpdf import FPDF

pdf = FPDF()
pdf.set_font("helvetica", size=24)
pdf.add_page()
pdf.cell(w=pdf.epw, txt="Welcome on first page!", align="C")
pdf.add_page()
link = pdf.add_link()
pdf.set_link(link, page=1)
pdf.cell(txt="Internal link to first page", border=1, link=link)
pdf.output("internal_link.pdf")
```

Similarly, `FPDF.link` can be used instead of `FPDF.cell`, however `write_html` does not allow to define internal links.

6.5 Alternative description

An optional textual description of the link can be provided, for accessibility purposes:

```
pdf.link(x=0, y=0, w=width, h=line_height, link="https://github.com/PyFPDF/fpdf2",
         alt_text="GitHub page for fpdf2")
```

7. Tables

Tables can be built either using **cells** or with `write_html`.

7.1 Using cells

There is a method to build tables allowing for multilines content in cells:

```
from fpdf import FPDF

data = (
    ("First name", "Last name", "Age", "City"),
    ("Jules", "Smith", "34", "San Juan"),
    ("Mary", "Ramos", "45", "Orlando"),
    ("Carlson", "Banks", "19", "Los Angeles"),
    ("Lucas", "Cimon", "31", "Saint-Mahturin-sur-Loire"),
)

pdf = FPDF()
pdf.add_page()
pdf.set_font("Times", size=10)
line_height = pdf.font_size * 2.5
col_width = pdf.epw / 4  # distribute content evenly
for row in data:
    for datum in row:
        pdf.multi_cell(col_width, line_height, datum, border=1, ln=3, max_line_height=pdf.font_size)
    pdf.ln(line_height)
pdf.output('table_with_cells.pdf')
```

7.2 Using write_html

An alternative method using `fpdf.HTMLMixin`, with the same `data` as above, and column widths defined as percent of the effective width:

```
from fpdf import FPDF, HTMLMixin

class PDF(FPDF, HTMLMixin):
    pass

pdf = PDF()
pdf.set_font_size(16)
pdf.add_page()
pdf.write_html(
    f"""<table border="1"><thead><tr>
    <th width="25%">{data[0][0]}</th>
    <th width="25%">{data[0][1]}</th>
    <th width="15%">{data[0][2]}</th>
    <th width="35%">{data[0][3]}</th>
</tr></thead><tbody><tr>
    <td>{'</td><td>'.join(data[1])}</td>
</tr><tr>
    <td>{'</td><td>'.join(data[2])}</td>
</tr><tr>
    <td>{'</td><td>'.join(data[3])}</td>
</tr><tr>
    <td>{'</td><td>'.join(data[4])}</td>
</tr></tbody></table>""",
    table_line_separators=True,
)
pdf.output('table_html.pdf')
```

7.3 Repeat table header on each page

The following recipe demonstrates a solution to handle this requirement:

```
from fpdf import FPDF

TABLE_COL_NAMES = ("First name", "Last name", "Age", "City")
TABLE_DATA = (
    ("Jules", "Smith", "34", "San Juan"),
    ("Mary", "Ramos", "45", "Orlando"),
    ("Carlson", "Banks", "19", "Los Angeles"),
    ("Lucas", "Cimon", "31", "Angers"),
)
```

```
pdf = FPDF()
pdf.add_page()
pdf.set_font("Times", size=16)
line_height = pdf.font_size * 2
col_width = pdf.epw / 4  # distribute content evenly

def render_table_header():
    pdf.set_font(style="B")  # enabling bold text
    for col_name in TABLE_COL_NAMES:
        pdf.cell(col_width, line_height, col_name, border=1)
    pdf.ln(line_height)
    pdf.set_font(style="")  # disabling bold text

render_table_header()
for _ in range(10):  # repeat data rows
    for row in TABLE_DATA:
        if pdf.will_page_break(line_height):
            render_table_header()
        for datum in row:
            pdf.cell(col_width, line_height, datum, border=1)
        pdf.ln(line_height)

pdf.output("table_with_headers_on_every_page.pdf")
```

Note that if you want to use `multi_cell()` method instead of `cell()`, some extra code will be required: an initial call to `multi_cell` with `split_only=True` will be needed in order to compute the number of lines in the cell.

8. Layout

8.1 Page format and orientation

By default, a `FPDF` document has a `A4` format with `portrait` orientation.

Other formats & orientation can be specified to `FPDF` constructor:

```
pdf = fpdf.FPDF(orientation="landscape", format="A5")
```

Currently supported formats are `a3`, `a4`, `a5`, `letter`, `legal` or a tuple `(width, height)`. Additional standard formats are welcome and can be suggested through pull requests.

8.1.1 Per-page format and orientation

The following code snippet illustrate how to configure different page formats for specific pages:

```
from fpdf import FPDF

pdf = FPDF()
pdf.set_font("Helvetica")
for i in range(9):
    pdf.add_page(format=(210 * (1 - i/10), 297 * (1 - i/10)))
    pdf.cell(txt=str(i))
pdf.add_page(same=True)
pdf.cell(txt="9")
pdf.output("varying_format.pdf")
```

Similarly, an `orientation` parameter can be provided to the `add_page` method.

8.2 Margins

By default a `FPDF` document has a 2cm margin at the bottom, and 1cm margin on the other sides.

Those margins control the initial current X & Y position to render elements on a page, and also define the height limit that triggers automatic page breaks when they are enabled.

Margins can be completely removed:

```
pdf.set_margin(0)
```

Several methods can be used to set margins:

- set_margin
- set_left_margin
- set_right_margin
- set_top_margin
- set_margins
- set_auto_page_break

8.3 Line breaks

When using multi_cell(), each time a line reaches the right extremity of the cell or a carriage return character (`\n`) is met, a line break is issued and a new cell automatically created under the current one.

An automatic break is performed at the location of the nearest space or soft-hyphen (`\u00ad`) character before the right limit. A soft-hyphen will be replaced by a normal hyphen when triggering a line break, and ignored otherwise.

8.4 Page breaks

By default, `fpdf2` will automatically perform page breaks whenever a cell or the text from a `write()` is rendered at the bottom of a page with a height greater than the page bottom margin.

This behaviour can be controlled using the `set_auto_page_break` and `accept_page_break` methods.

8.4.1 Manually trigger a page break

Simply call `.add_page()`.

8.4.2 Unbreakable sections

In order to render content, like tables, with the insurance that no page break will be performed in it, on the can use the `FPDF.unbreakable()` context-manager:

```
pdf = fpdf.FPDF()
pdf.add_page()
pdf.set_font("Times", size=16)
line_height = pdf.font_size * 2
col_width = pdf.epw / 4  # distribute content evenly
for i in range(4):  # repeat table 4 times
    with pdf.unbreakable() as pdf:
        for row in data:  # data comes from snippets on the Tables documentation page
            for datum in row:
                pdf.cell(col_width, line_height, f"{datum} ({i})", border=1)
            pdf.ln(line_height)
    pdf.ln(line_height * 2)
pdf.("unbreakable_tables.pdf")
```

9. Document outline & table of contents

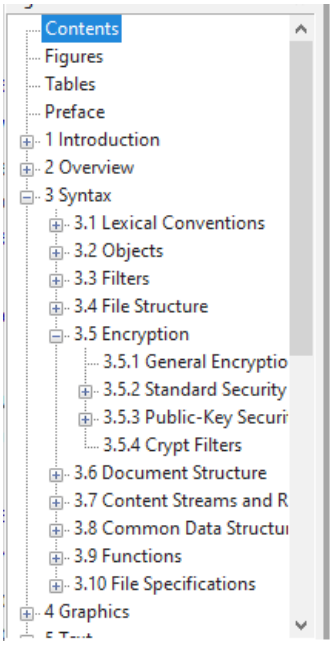
Quoting Wikipedia, a **table of contents** is:

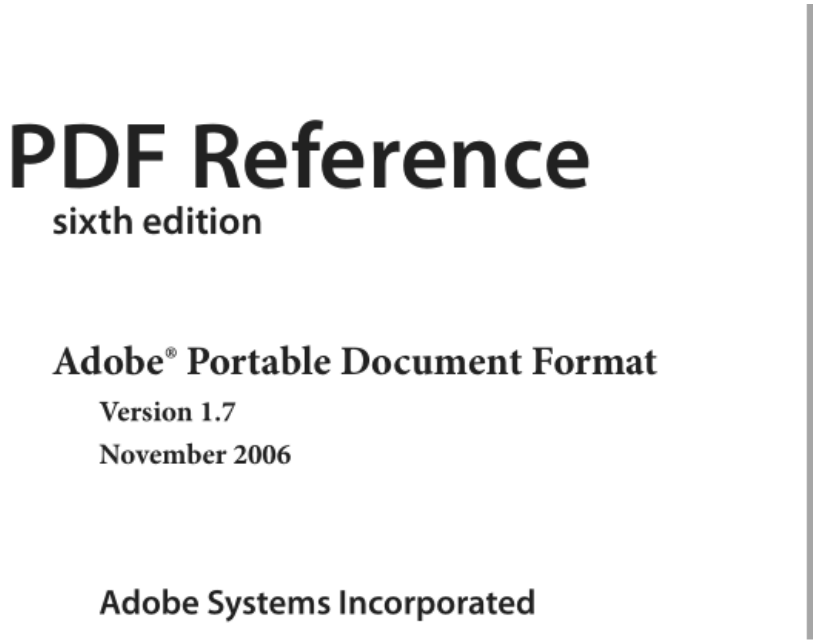
a list, usually found on a page before the start of a written work, of its chapter or section titles or brief descriptions with their commencing page numbers.

Now quoting the 6th edition of the PDF format reference (v1.7 - 2006) :

A PDF document may optionally display a **document outline** on the screen, allowing the user to navigate interactively from one part of the document to another. The outline consists of a tree-structured hierarchy of outline items (sometimes called bookmarks), which serve as a visual table of contents to display the document's structure to the user.

For example, there is how a document outline looks like in Sumatra PDF Reader:





Since `fpdf2.3.3`, both features are supported through the use of the `start_section` method, that adds an entry in the internal "outline" table used to render both features.

Note that by default, calling `start_section` only records the current position in the PDF and renders nothing. However, you can configure **global title styles** by calling `set_section_title_styles`, after which call to `start_section` will render titles visually using the styles defined.

To provide a document outline to the PDF you generate, you just have to call the `start_section` method for every hierarchical section you want to define.

If you also want to insert a table of contents somewhere, call `insert_toc_placeholder` wherever you want to put it. Note that a page break will always be triggered after inserting the table of contents.

9.1 With HTML

When using `fpdf.HTMLMixin`, a document outline is automatically built. You can insert a table of content with the special `<toc>` tag.

Custom styling of the table of contents can be achieved by overriding the `render_toc` method in a subclass of `fpdf.html.HTML2FPDF`:

```
from fpdf import FPDF, HTMLMixin, HTML2FPDF

class CustomHTML2FPDF(HTML2FPDF):
    def render_toc(self, pdf, outline):
        pdf.cell(txt='Table of contents:', ln=1)
        for section in outline:
            pdf.cell(txt=f'* {section.name} (page {section.page_number})', ln=1)

class PDF(FPDF, HTMLMixin):
    HTML2FPDF_CLASS = CustomHTML2FPDF

pdf = PDF()
pdf.add_page()
pdf.write_html("""<toc></toc>
    <h1>Level 1</h1>
    <h2>Level 2</h2>
    <h3>Level 3</h3>
    <h4>Level 4</h4>
    <h5>Level 5</h5>
    <h6>Level 6</h6>
    <p>paragraph<p>""")
pdf.output("html_toc.pdf")
```

9.2 Code samples

The regression tests are a good place to find code samples.

For example, the `test_simple_outline` test function generates the PDF document test_simple_outline.pdf.

Similarly, `test_html_toc` generates test_html_toc.pdf.

10. Text styling

`fpdf2` supports setting *emphasis* on text : **bold**, **italics** or underlined.

Bold & italics require using dedicated fonts for each style.

For the standard fonts (Courier, Helvetica & Times), those dedicated fonts are configured by default. Using other fonts means that their variants (bold, italics) must be registered using `add_font` (with `style="B"` and `style="I"`).

10.1 set_font

Setting emphasis on text can be controlled by using `set_font(style=...)`

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font("Times", size=36)
pdf.cell(txt="This")
pdf.set_font(style="B")
pdf.cell(txt="is")
pdf.set_font(style="I")
pdf.cell(txt="a")
pdf.set_font(style="U")
pdf.cell(txt="PDF")
pdf.output("style.pdf")
```

10.2 write_html

`write_html` allows to set emphasis on text through the `<b>`, `<i>` and `<u>` tags:

```
pdf.write_html("""<B>bold</B>
                  <I>italic</I>
                  <U>underlined</U>
                  <B><I><U>all at once!</U></I></B>"""
)
```

10.3 markdown=True

An optional `markdown=True` parameter can be passed to the `cell` method in order to enable basic Markdown-like styling: `**bold**, __italics__, --underlined--`

```
from fpdf import FPDF

pdf = fpdf.FPDF()
pdf.add_page()
pdf.set_font("Times", size=60)
pdf.cell(txt="**Lorem** __Ipsum__ --dolor--", markdown=True)
pdf.output("markdown-styled.pdf")
```

11. Presentations

Presentation mode can usually be enabled with the `CTRL + L` shortcut.

As of june 2021, the features described below are onored by Adobe Acrobat reader, but ignored by Sumatra PDF reader.

11.1 Page display duration

Pages can be associated with a "display duration" until when the viewer application automatically advances to the next page:

```
from fpdf import FPDF

pdf = fpdf.FPDF()
pdf.set_font("Helvetica", size=120)
pdf.add_page(duration=3)
pdf.cell(txt="Page 1")
pdf.page_duration = .5
pdf.add_page()
pdf.cell(txt="Page 2")
pdf.add_page()
pdf.cell(txt="Page 3")
pdf.output("presentation.pdf")
```

It can also be configured globally through the `page_duration` FPDF property.

11.2 Transitions

Pages can be associated with visual transitions to use when moving from another page to the given page during a presentation:

```
from fpdf import FPDF
from fpdf.transitions import *

pdf = fpdf.FPDF()
pdf.set_font("Helvetica", size=120)
pdf.add_page()
pdf.text(x=40, y=150, txt="Page 0")
pdf.add_page(transition=SplitTransition("V", "O"))
pdf.text(x=40, y=150, txt="Page 1")
pdf.add_page(transition=BlindsTransition("H"))
pdf.text(x=40, y=150, txt="Page 2")
pdf.add_page(transition=BoxTransition("I"))
pdf.text(x=40, y=150, txt="Page 3")
pdf.add_page(transition=WipeTransition(90))
pdf.text(x=40, y=150, txt="Page 4")
pdf.add_page(transition=DissolveTransition())
pdf.text(x=40, y=150, txt="Page 5")
pdf.add_page(transition=GlitterTransition(315))
pdf.text(x=40, y=150, txt="Page 6")
pdf.add_page(transition=FlyTransition("H"))
pdf.text(x=40, y=150, txt="Page 7")
pdf.add_page(transition=PushTransition(270))
pdf.text(x=40, y=150, txt="Page 8")
pdf.add_page(transition=CoverTransition(270))
pdf.text(x=40, y=150, txt="Page 9")
pdf.add_page(transition=UncoverTransition(270))
pdf.text(x=40, y=150, txt="Page 10")
pdf.add_page(transition=FadeTransition())
pdf.text(x=40, y=150, txt="Page 11")
pdf.output("transitions.pdf")
```

It can also be configured globally through the `page_transition` FPDF property.

12. Annotations

The PDF format allows to add various annotations to a document.

12.1 Text annotations

They are rendered this way by Sumatra PDF reader:


```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font("Helvetica", size=24)
pdf.text(x=60, y=140, txt="Some text.")
pdf.text_annotation(
    x=100,
    y=130,
    text="This is a text annotation.",
)
pdf.output("text_annotation.pdf")
```

12.2 Named actions

The four standard PDF named actions provide some basic navigation relative to the current page: `NextPage`, `PrevPage`, `FirstPage` and `LastPage`.

```
from fpdf import FPDF
from fpdf.actions import NamedAction

pdf = FPDF()
pdf.set_font("Helvetica", size=24)
pdf.add_page()
pdf.text(x=80, y=140, txt="First page")
pdf.add_page()
pdf.underline = True
for x, y, named_action in ((40, 80, "NextPage"), (120, 80, "PrevPage"), (40, 200, "FirstPage"), (120, 200, "LastPage")):
    pdf.text(x=x, y=y, txt=named_action)
    pdf.add_action(
        NamedAction(named_action),
        x=x,
        y=y - pdf.font_size,
        w=pdf.get_string_width(named_action),
        h=pdf.font_size,
    )
pdf.underline = False
pdf.add_page()
pdf.text(x=80, y=140, txt="Last page")
pdf.output("named_actions.pdf")
```

12.3 Launch actions

Used to launch an application or open or print a document:

```
from fpdf import FPDF
from fpdf.actions import LaunchAction

pdf = FPDF()
pdf.set_font("Helvetica", size=24)
pdf.add_page()
x, y, text = 80, 140, "Launch action"
pdf.text(x=x, y=y, txt=text)
pdf.add_action(
    LaunchAction("another_file_in_same_directory.pdf"),
    x=x,
    y=y - pdf.font_size,
    w=pdf.get_string_width(text),
```

```
    h=pdf.font_size,
)
pdf.output("launch_action.pdf")
```

13. Shapes

The following code snippets show examples of rendering various shapes.

13.1 Lines

Draw a thin plain orange line:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_line_width(0.5)
pdf.set_draw_color(r=255, g=128, b=0)
pdf.line(x1=50, y1=50, x2=150, y2=100)
pdf.output("orange_plain_line.pdf")
```

Draw a dashed light blue line:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_line_width(0.5)
pdf.set_draw_color(r=0, g=128, b=255)
pdf.set_dash_pattern(dash=2, gap=3)
pdf.line(x1=50, y1=50, x2=150, y2=100)
pdf.output("blue_dashed_line.pdf")
```

13.2 Ellipse

Draw a circle filled in grey with a pink outline:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_line_width(2)
pdf.set_draw_color(r=230, g=30, b=180)
pdf.set_fill_color(240)
pdf.ellipse(x=50, y=50, w=50, h=50, style="FD")
pdf.output("circle.pdf")
```

13.3 Rectangle

Draw nested squares:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
for i in range(15):
    pdf.set_fill_color(255 - 15*i)
    pdf.rect(x=5+5*i, y=5+5*i, w=200-10*i, h=200-10*i, style="FD")
pdf.output("squares.pdf")
```

13.4 Polygon

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_line_width(2)
pdf.set_fill_color(r=255, g=0, b=0)
coords = ((100, 0), (5, 69), (41, 181), (159, 181), (195, 69))
pdf.polygon(coords, fill=True)
pdf.output("polygon.pdf")
```

13.5 Arc

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_line_width(2)
pdf.set_fill_color(r=255, g=0, b=0)
pdf.arc(x=75, y=75, a=25, b=25, start_angle=30, end_angle=130, style="FD")
pdf.output("arc.pdf")
```

13.6 Solid arc

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_line_width(2)
pdf.set_fill_color(r=255, g=0, b=0)
pdf.solid_arc(x=75, y=75, a=25, b=25, start_angle=30, end_angle=130, style="FD")
pdf.output("solid_arc.pdf")
```

14. Metadata

`fpdf2` can only insert mathematical formula in the form of **images**. The following sections will explaing how to generate and embed such images.

14.1 Using Google Charts API

Official documentation: Google Charts Infographics - Mathematical Formulas. Example:

```
from io import BytesIO
from urllib.parse import quote
from urllib.request import urlopen
from fpdf import FPDF

formula = 'x^n + y^n = a/b'
height = 170
url = f"https://chart.googleapis.com/chart?cht=tx&chs={height}&chl={quote(formula)}"
with urlopen(url) as img_file:
    img = BytesIO(img_file.read())

pdf = FPDF()
pdf.add_page()
pdf.image(img, w=30)
pdf.output("equation-with-gcharts.pdf")
```

14.2 Using Matplotlib

Example:

```
from fpdf import FPDF
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure
import numpy
from PIL import Image


fig = Figure(figsize=(6, 2), dpi=100)
canvas = FigureCanvas(fig)
axes = fig.gca()
axes.text(0, .5, r"$x^n + y^n = \frac{a}{b}$", fontsize=60)  # LaTeX syntax
axes.axis("off")
canvas.draw()
img = Image.fromarray(numpy.asarray(canvas.buffer_rgba()))

pdf = FPDF()
pdf.add_page()
pdf.image(img, w=30)
pdf.output("equation-with-matplotlib.pdf")
```

15. Drawing

The `fpdf.drawing` module provides an API for composing paths out of an arbitrary sequence of straight lines and curves. This allows fairly low-level control over the graphics primitives that PDF provides, giving the user the ability to draw pretty much any vector shape on the page.

The drawing API makes use of features (notably transparency and blending modes) that were introduced in PDF 1.4. Therefore, use of the features of this module will automatically set the output version to 1.4 (fpdf normally defaults to version 1.3. Because the PDF 1.4 specification was originally published in 2001, this version should be compatible with all viewers currently in general use).

15.1 Getting Started

The easiest way to add a drawing to the document is via `fpdf.FPDF.new_path`. This is a context manager that takes care of serializing the path to the document once the context is exited.

Drawings follow the fpdf convention that the origin (that is, coordinate(0, 0)), is at the top-left corner of the page. The numbers specified to the various path commands are interpreted in the document units.

```
import fpdf

pdf = fpdf.FPDF(unit='mm', format=(10, 10))
pdf.add_page()

with pdf.new_path() as path:
    path.move_to(2, 2)
    path.line_to(8, 8)
    path.horizontal_line_relative(-6)
    path.line_relative(6, -6)
    path.close()

pdf.output("drawing-demo.pdf")
```

This example draws an hourglass shape centered on the page:

view as PDF

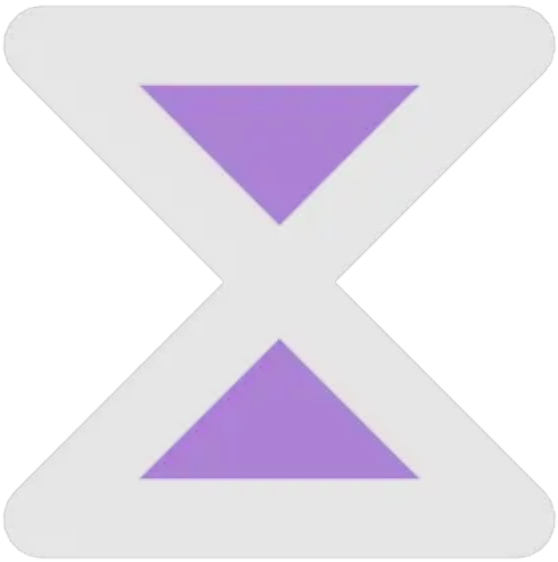
15.2 Adding Some Style

Drawings can be styled, changing how they look and blend with other drawings. Styling can change the color, opacity, stroke shape, and other attributes of a drawing.

Let's add some color to the above example:

```
import fpdf

pdf = fpdf.FPDF(unit='mm', format=(10, 10))
pdf.add_page()

with pdf.new_path() as path:
    path.style.fill_color = '#A070D0'
    path.style.stroke_color = fpdf.drawing.gray8(210)
    path.style.stroke_width = 1
    path.style.stroke_opacity = 0.75
    path.style.stroke_join_style = 'round'

    path.move_to(2, 2)
    path.line_to(8, 8)
    path.horizontal_line_relative(-6)
    path.line_relative(6, -6)
    path.close()

pdf.output("drawing-demo.pdf")
```

If you make color choices like these, it's probably not a good idea to quit your day job to become a graphic designer. Here's what the output should look like:



view as PDF

15.3 Transforms And You

Transforms provide the ability to manipulate the placement of points within a path without having to do any pesky math yourself. Transforms are composable using python's matrix multiplication operator (`@`), so, for example, a transform that both rotates and scales an object can be create by matrix multiplying a rotation transform with a scaling transform.

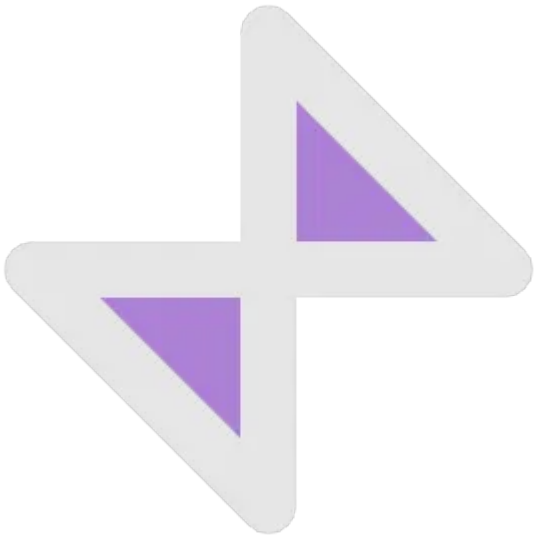
An important thing to note about transforms is that the result is order dependent, which is to say that something like performing a rotation followed by scaling will not, in the general case, result in the same output as performing the same scaling followed by the same rotation.

Additionally, it's not generally possible to deconstruct a composed transformation (representing an ordered sequence of translations, scaling, rotations, shearing) back into the sequence of individual transformation functions that produced it. That's okay, because this isn't important unless you're trying to do something like animate transforms after they've been composed, which you can't do in a PDF anyway.

All that said, let's take the example we've been working with for a spin (the pun is intended, you see, because we're going to rotate the drawing). Explaining the joke does make it better.

An easy way to apply a transform to a path is through the `path.transform` property.

```
import fpdf

pdf = fpdf.FPDF(unit="mm", format=(10, 10))
pdf.add_page()

with pdf.new_path() as path:
    path.style.fill_color = "#A070D0"
    path.style.stroke_color = fpdf.drawing.gray8(210)
    path.style.stroke_width = 1
    path.style.stroke_opacity = 0.75
    path.style.stroke_join_style = "round"
    path.transform = fpdf.drawing.Transform.rotation_d(45).scale(0.707).about(5, 5)

    path.move_to(2, 2)
    path.line_to(8, 8)
    path.horizontal_line_relative(-6)
    path.line_relative(6, -6)

    path.close()

pdf.output("drawing-demo.pdf")
```



view as PDF

The transform in the above example rotates the path 45 degrees clockwise and scales it by `1/sqrt(2)` around its center point. This transform could be equivalently written as:

```
import fpdf
T = fpdf.drawing.Transform

T.translation(-5, -5) @ T.rotation_d(45) @ T.scaling(0.707) @ T.translation(5, 5)
```

Because all transforms operate on points relative to the origin, if we had rotated the path without first centering it on the origin, we would have rotated it partway off of the page. Similarly, the size-reduction from the scaling would have moved it closer to the origin. By bracketing the transforms with the two translations, the placement of the drawing on the page is preserved.

15.4 Clipping Paths

The clipping path is used to define the region that the normal path is actually painted. This can be used to create drawings that would otherwise be difficult to produce.

```
import fpdf

pdf = fpdf.FPDF(unit="mm", format=(10, 10))
pdf.add_page()

clipping_path = fpdf.drawing.ClippingPath()
clipping_path.rectangle(x=2.5, y=2.5, w=5, h=5, rx=1, ry=1)

with pdf.new_path() as path:
    path.style.fill_color = "#A070D0"
    path.style.stroke_color = fpdf.drawing.gray8(210)
    path.style.stroke_width = 1
    path.style.stroke_opacity = 0.75
    path.style.stroke_join_style = "round"

    path.clipping_path = clipping_path

    path.move_to(2, 2)
    path.line_to(8, 8)
    path.horizontal_line_relative(-6)
    path.line_relative(6, -6)

    path.close()

pdf.output("drawing-demo.pdf")
```

view as PDF

15.5 Next Steps

The presented API style is designed to make it simple to produce shapes declaratively in your Python scripts. However, paths can just as easily be created programmatically by creating instances of the `fpdf.drawing.PaintedPath` for paths and `fpdf.drawing.GraphicsContext` for groups of paths.

Storing paths in intermediate objects allows reusing them and can open up more advanced use-cases. The `fpdf.svg` SVG converter, for example, is implemented using the `fpdf.drawing` interface.

16. Scalable Vector Graphics (SVG)

`fpdf2` supports basic conversion of SVG paths into PDF paths, which can be inserted into an existing PDF document or used as the contents of a new PDF document.

Not all SVGs will convert correctly. Please see the list of unsupported features for more information about what to look out for.

16.1 A simple example

The following script will create a PDF that consists only of the graphics contents of the provided SVG file:

```
import fpdf

svg = fpdf.svg.SVGObject.from_file("my_file.svg")

pdf = fpdf.FPDF(unit="pt", format=(svg.width, svg.height))
pdf.add_page()
svg.draw_to_page(pdf)

pdf.output("my_file.pdf")
```

Because this takes the PDF document size from the source SVG, it does assume that the width/height of the SVG are specified in absolute units rather than relative ones (i.e. the top-level `<svg>` tag has something like `width="5cm"` and not `width=50%`). In this case, if the values are percentages, they will be interpreted as their literal numeric value (i.e. `100%` would be treated as `100 pt`). The next example uses `transform_to_page_viewport`, which will scale an SVG with a percentage based `width` to the pre-defined PDF page size.

The converted SVG object can be returned as an fpdf.drawing.GraphicsContext collection of drawing directives for more control over how it is rendered:

```
import fpdf

svg = fpdf.svg.SVGObject.from_file("my_file.svg")

pdf = FPDF(unit="in", format=(8.5, 11))
pdf.add_page()

# We pass align_viewbox=False because we want to perform positioning manually
# after the size transform has been computed.
width, height, paths = svg.transform_to_page_viewport(pdf, align_viewbox=False)
# note: transformation order is important! This centers the svg drawing at the
# origin, rotates it 90 degrees clockwise, and then repositions it to the
# middle of the output page.
paths.transform = paths.transform @ fpdf.drawing.Transform.translation(
    -width / 2, -height / 2
).rotate_d(90).translate(pdf.w / 2, pdf.h / 2)

pdf.draw_path(paths)

pdf.output("my_file.pdf")
```

16.2 Supported SVG Features

- groups
- paths
- basic shapes (rect, circle, ellipse, line, polyline, polygon)
- basic cross-references
- stroke & fill coloring and opacity
- basic stroke styling

16.3 Currently Unsupported Notable SVG Features

Everything not listed as supported is unsupported, which is a lot. SVG is a ridiculously complex format that has become increasingly complex as it absorbs more of the entire browser rendering stack into its specification. However, there are some pretty commonly used features that are unsupported that may cause unexpected results (up to and including a normal-looking SVG rendering as a completely blank PDF). It is very likely that off-the-shelf SVGs will not be converted fully correctly without some preprocessing.

The biggest unsupported feature is probably:

- CSS styling of SVG elements

In addition to that:

- text/tspan/textPath
- symbols
- markers
- patterns
- gradients
- embedded images or other content (including nested SVGs)
- a lot of attributes

17. Barcodes

17.1 Code 39

Here is an example on how to generate a Code 39 barcode:

```
pdf = FPDF()
pdf.add_page()
pdf.code39("*fpdf2*", x=30, y=50, w=4, h=20)
pdf.output("code39.pdf")
```

Output preview:

17.2 Interleaved 2 of 5

Here is an example on how to generate an Interleaved 2 of 5 barcode:

```
pdf = FPDF()
pdf.add_page()
pdf.interleaved2of5("1337", x=50, y=50, w=4, h=20)
pdf.output("interleaved2of5.pdf")
```

Output preview:

17.3 PDF-417

Here is an example on how to generate a PDF-417 barcode using the `pdf417` lib:

```
from pdf417 import encode, render_image

pdf = FPDF()
pdf.add_page()
img = render_image(encode("Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed non risus. Suspendisse lectus tortor, dignissim sit amet, adipiscing
```

```
nec, ultricies sed, dolor. Cras elementum ultrices diam."))
pdf.image(img, x=10, y=50)
pdf.output("pdf417.pdf")
```

Output preview:

17.4 QRCode

Here is an example on how to generate a QR Code using the `python-qrcode` lib:

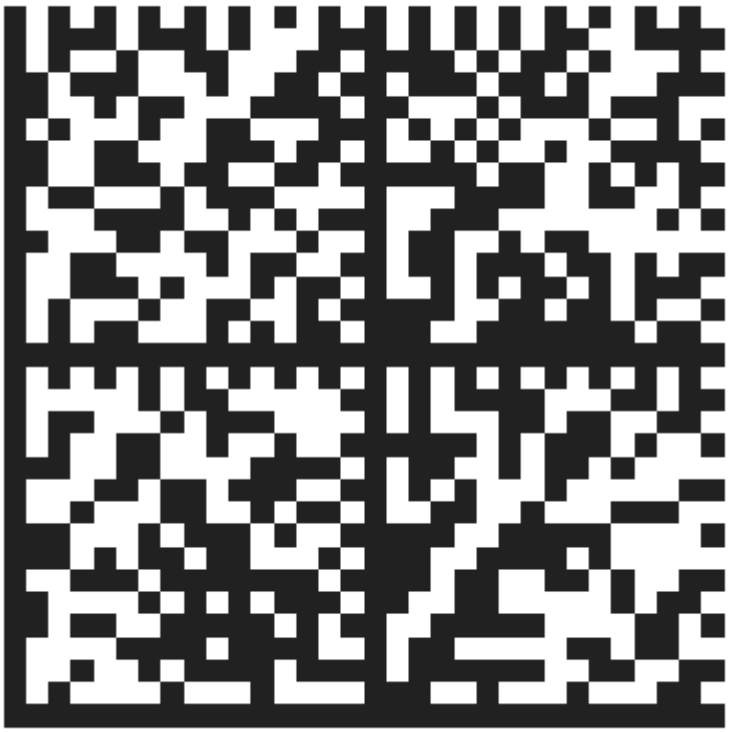
```
import qrcode

pdf = FPDF()
pdf.add_page()
img = qrcode.make("fpdf2")
pdf.image(img.get_image(), x=50, y=50)
pdf.output("qrcode.pdf")
```

Output preview:

17.5 DataMatrix

`fpdf2` can be combined with the `pystrich` library to generate DataMatrix barcodes: `pystrich` generates pilimages, which can then be inserted into the PDF file via the `FPDF.image()` method.

```
from fpdf import FPDF
from pystrich.datamatrix import DataMatrixEncoder, DataMatrixRenderer


# Define the properties of the barcode
positionX = 10
positionY = 10
width = 57
height = 57
cellsize = 5

# Prepare the datamatrix renderer that will be used to generate the pilimage
encoder = DataMatrixEncoder("[Text to be converted to a datamatrix barcode]")
encoder.width = width
encoder.height = height
renderer = DataMatrixRenderer(encoder.matrix, encoder.regions)

# Generate a pilimage and move it into the memory stream
img = renderer.get_pilimage(cellsize)

# Draw the barcode image into a PDF file
pdf = FPDF()
pdf.add_page()
pdf.image(img, positionX, positionY, width, height)
```



17.5.1 Extend FPDF with a datamatrix() method

The code above could be added to the FPDF class as an extension method in the following way.

```
from fpdf import FPDF
from pystrich.datamatrix import DataMatrixEncoder, DataMatrixRenderer


def datamatrix(self, text='', x=0, y=0, w=57, h=57, cellsize=5):
    "Convert text to a datamatrix barcode and put in on the page"
    encoder = DataMatrixEncoder(text)
```

```
    encoder.width = w
    encoder.height = h
    renderer = DataMatrixRenderer(encoder.matrix, encoder.regions)
    img = renderer.get_pilimage(cellsize)
    self.image(img, x, y, w, h)

FPDF.datamatrix = datamatrix
```

18. Metadata

The PDF specification contain two types of metadata, the newer XMP (Extensible Metadata Platform, XML-based) and older `DocumentInformation` dictionary. The PDF 2.0 specification removes the `DocumentInformation` dictionary.

Currently, the following methods on `fpdf.FPDF` allow to set metadata information in the `DocumentInformation` dictionary:

- `set_title`
- `set_lang`
- `set_subject`
- `set_author`
- `set_keywords`
- `set_producer`
- `set_creator`
- `set_creation_date`
- `set_xmp_metadata`, that requires you to craft the necessary XML string

For a more user-friendly API to set metadata, we recommend using `pikepdf` that will set both XMP & `DocumentInformation` metadata:

```
import sys
from datetime import datetime

import pikepdf
from fpdf import FPDF_VERSION

with pikepdf.open(sys.argv[1], allow_overwriting_input=True) as pdf:
    with pdf.open_metadata(set_pikepdf_as_editor=False) as meta:
        meta["dc:title"] = "Title"
        meta["dc:description"] = "Description"
        meta["dc:creator"] = ["Author1", "Author2"]
        meta["pdf:Keywords"] = "keyword1 keyword2 keyword3"
        meta["pdf:Producer"] = f"PyFPDF/fpdf{FPDF_VERSION}"
        meta["xmp:CreatorTool"] = __file__
        meta["xmp:MetadataDate"] = datetime.now(datetime.utcnow().astimezone().tzinfo).isoformat()
    pdf.save()
```

19. Logging

`fpdf.FPDF` generates useful `DEBUG` logs on generated sections sizes when calling the `output()` method., that can help to identify what part of a PDF takes most space (fonts, images, pages...).

Here is an example of setup code to display them:

```
import logging

logging.basicConfig(format="%(asctime)s %(filename)s [%(levelname)s] %(message)s",
                    datefmt="%H:%M:%S", level=logging.DEBUG)
```

Example output using the Tutorial first code snippet:

```
14:09:56 fpdf.py [DEBUG] Final doc sections size summary:
14:09:56 fpdf.py [DEBUG] - header.size: 9.0B
14:09:56 fpdf.py [DEBUG] - pages.size: 306.0B
14:09:56 fpdf.py [DEBUG] - resources.fonts.size: 101.0B
14:09:56 fpdf.py [DEBUG] - resources.images.size: 0.0B
14:09:56 fpdf.py [DEBUG] - resources.dict.size: 104.0B
14:09:56 fpdf.py [DEBUG] - info.size: 54.0B
14:09:56 fpdf.py [DEBUG] - catalog.size: 103.0B
14:09:56 fpdf.py [DEBUG] - xref.size: 169.0B
14:09:56 fpdf.py [DEBUG] - trailer.size: 60.0B
```

20. Templates

Templates are predefined documents (like invoices, tax forms, etc.), or parts of such documents, where each element (text, lines, barcodes, etc.) has a fixed position (x1, y1, x2, y2), style (font, size, etc.) and a default text.

These elements can act as placeholders, so the program can change the default text "filling in" the document.

Besides being defined in code, the elements can also be defined in a CSV file or in a database, so the user can easily adapt the form to his printing needs.

A template is used like a dict, setting its items' values.

How to use Templates?

There are two approaches to using templates, detailed in the sections below:

20.1 Using Template()

The traditional approach is to use the Template() class, This class accepts one template definition, and can apply it to each page of a document. The usage pattern here is:

```
tmpl = Template(elements=elements)
# first page and content
tmpl.add_page()
tmpl[item_key_01] = "Text 01"
tmpl[item_key_02] = "Text 02"
...

# second page and content
tmpl.add_page()
tmpl[item_key_01] = "Text 11"
tmpl[item_key_02] = "Text 12"
...

# possibly more pages
...

# finalize document and write to file
tmpl.render(outfile="example.pdf")
```

The Template() class will create and manage its own FPDF() instance, so you don't need to worry about how it all works together. It also allows to set the page format, title of the document, measuring unit, and other metadata for the PDF file.

For the method signatures, see pyfpdf.github.io: class Template.

Setting text values for specific template items is done by treating the class as a dict, with the name of the item as the key:

```
Template["company_name"] = "Sample Company"
```

20.2 Using FlexTemplate()

When more flexibility is desired, then the FlexTemplate() class comes into play. In this case, you first need to create your own FPDF() instance. You can then pass this to the constructor of one or several FlexTemplate() instances, and have each of them load a template definition. For any page of the document, you can set text values on a template, and then render it on that page. After rendering, the template will be reset to its default values.

```
pdf = FPDF()
pdf.add_page()
# One template for the first page
fp_tmpl = FlexTemplate(pdf, elements=fp_elements)
fp_tmpl["item_key_01"] = "Text 01"
fp_tmpl["item_key_02"] = "Text 02"
...
fp_tmpl.render() # add template items to first page

# add some more non-template content to the first page
pdf.polyline(point_list, fill=False, polygon=False)
```

```
# second page
pdf.add_page()
# header for the second page
h_tmpl = FlexTemplate(pdf, elements=h_elements)
h_tmpl["item_key_HA"] = "Text 2A"
h_tmpl["item_key_HB"] = "Text 2B"
...
h_tmpl.render() # add header items to second page

# footer for the second page
f_tmpl = FlexTemplate(pdf, elements=f_elements)
f_tmpl["item_key_FC"] = "Text 2C"
f_tmpl["item_key_FD"] = "Text 2D"
...
f_tmpl.render() # add footer items to second page

# other content on the second page
pdf.set_dash_pattern(dash=1, gap=1)
pdf.line(x1, y1, x2, y2):
pdf.set_dash_pattern()

# third page
pdf.add_page()
# header for the third page, just reuse the same template instance after render()
h_tmpl["item_key_HA"] = "Text 3A"
h_tmpl["item_key_HB"] = "Text 3B"
...
h_tmpl.render() # add header items to third page

# footer for the third page
f_tmpl["item_key_FC"] = "Text 3C"
f_tmpl["item_key_FD"] = "Text 3D"
...
f_tmpl.render() # add footer items to third page

# other content on the third page
pdf.rect(x, y, w, h, style=None)

# possibly more pages
pdf.next_page()
...
...

# finally write everything to a file
pdf.output("example.pdf")
```

Evidently, this can end up quite a bit more involved, but there are hardly any limits on how you can combine templated and non-templated content on each page. Just think of the different templates as of building blocks, like configurable rubber stamps, which you can apply in any combination on any page you like.

Of course, you can just as well use a set of full-page templates, possibly differentiating between cover page, table of contents, normal content pages, and an index page, or something along those lines.

And here's how you can use a template several times on one page (and by extension, several times on several pages). When rendering with an `offsetx` and/or `offsety` argument, the contents of the template will end up in a different place on the page. A `rotate` argument will change its orientation, rotated around the origin of the template. The pivot of the rotation is the offset location. And finally, a `scale` argument allows you to insert the template larger or smaller than it was defined.

```
elements = [
    {"name":"box", "type":"B", "x1":0, "y1":0, "x2":50, "y2":50,},
    {"name":"d1", "type":"L", "x1":0, "y1":0, "x2":50, "y2":50,},
    {"name":"d2", "type":"L", "x1":0, "y1":50, "x2":50, "y2":0,},
    {"name":"label", "type":"T", "x1":0, "y1":52, "x2":50, "y2":57, "text":"Label",},
]
pdf = FPDF()
pdf.add_page()
templ = FlexTemplate(pdf, elements)
templ["label"] = "Offset: 50 / 50 mm"
templ.render(offsetx=50, offsety=50)
templ["label"] = "Offset: 50 / 120 mm"
templ.render(offsetx=50, offsety=120)
templ["label"] = "Offset: 120 / 50 mm, Scale: 0.5"
templ.render(offsetx=120, offsety=50, scale=0.5)
templ["label"] = "Offset: 120 / 120 mm, Rotate: 30°, Scale=0.5"
templ.render(offsetx=120, offsety=120, rotate=30.0, scale=0.5)
pdf.output("example.pdf")
```

For the method signatures, see pyfpdf.github.io: class FlexTemplate.

The dict syntax for setting text values is the same as above:

```
FlexTemplate["company_name"] = "Sample Company"
```

20.2.1 Details - Template definition

A template definition consists of a number of elements, which have the following properties (columns in a CSV, items in a dict, fields in a database). Dimensions (except font size, which always uses points) are given in user defined units (default: mm). Those are the units that can be specified when creating a `Template()` or a `FPDF()` instance.

- **name**: placeholder identification (unique text string)
- *mandatory*
- **type**:
- '**T**': Text - places one or several lines of text on the page
- '**L**': Line - draws a line from x1/y1 to x2/y2
- '**I**': Image - positions and scales an image into the bounding box
- '**B**': Box - draws a rectangle around the bounding box
- '**E**': Ellipse - draws an ellipse inside the bounding box
- '**BC**': Barcode - inserts an "Interleaved 2 of 5" type barcode
- '**C39**': Code 39 - inserts a "Code 39" type barcode
- Incompatible change: A previous implementation of this type used the non-standard element keys "x", "y", "w", and "h", which are now deprecated (but still work for the moment).
- '**W**': "Write" - uses the `FPDF.write()` method to add text to the page
- *mandatory*
- **x1, y1, x2, y2**: top-left, bottom-right coordinates, defining a bounding box in most cases
- for multiline text, this is the bounding box of just the first line, not the complete box
- for the barcodes types, the height of the barcode is `y2 - y1`, x2 is ignored.
- *mandatory* ("x2" *optional* for the barcode types)
- **font**: the name of a font type for the text types
- *optional*
- default: "helvetica"
- **size**: the size property of the element (float value)
- for text, the font size (in points!)
- for line, box, and ellipse, the line width
- for the barcode types, the width of one bar
- *optional*
- default: 10 for text, 2 for 'BC', 1.5 for 'C39'
- **bold, italic, underline**: text style properties
- in elements dict, enabled with True or equivalent value
- in csv, only int values, 0 as false, non-0 as true
- *optional*
- default: false
- **foreground, background**: text and fill colors (int value, commonly given in hex as 0xRRGGBB)
- *optional*
- default: foreground 0x000000 = black; background None/empty = transparent
- Incompatible change: Up to 2.4.5, the default background for text and box elements was solid white, with no way to make them transparent.
- **align**: text alignment, '**L**': left, '**R**': right, '**C**': center
- *optional*
- default: 'L'

- **text**: default string, can be replaced at runtime
- displayed text for 'T' and 'W'
- data to encode for barcode types
- *optional* (if missing for text/write, the element is ignored)
- default: empty
- **priority**: Z-order (int value)
- *optional*
- default: 0
- **multiline**: configure text wrapping
- in dicts, None for single line, True for multicells (multiple lines), False trims to exactly fit the space defined
- in csv, 0 for single line, >0 for multiple lines, <0 for exact fit
- *optional*
- default: single line
- **rotation**: rotate the element in degrees around the top left corner x1/y1 (float)
- *optional*
- default: 0.0 - no rotation

Fields that are not relevant to a specific element type will be ignored there, but if not left empty, they must still adhere to the specified data type (in dicts, string fields may be None).

20.2.2 How to create a template

A template can be created in 3 ways:

- By defining everything manually in a hardcoded way as a Python dictionary
- By using a template definition in a CSV document and parsing the CSV with Template.parse_dict()
- By defining the template in a database (this applies to [Web2Py] (Web2Py.md) integration)

20.2.3 Example - Hardcoded

```
from fpdf import Template

#this will define the ELEMENTS that will compose the template.
elements = [
    { 'name': 'company_logo', 'type': 'I', 'x1': 20.0, 'y1': 17.0, 'x2': 78.0, 'y2': 30.0, 'font': None, 'size': 0.0, 'bold': 0, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0, 'align': 'I', 'text': 'logo', 'priority': 2, 'multiline': 0},
    { 'name': 'company_name', 'type': 'T', 'x1': 17.0, 'y1': 32.5, 'x2': 115.0, 'y2': 37.5, 'font': 'helvetica', 'size': 12.0, 'bold': 1, 'italic': 0,
'underline': 0, 'foreground': 0, 'background': 0, 'align': 'I', 'text': '', 'priority': 2, 'multiline': 0},
    { 'name': 'multline_text', 'type': 'T', 'x1': 20, 'y1': 100, 'x2': 40, 'y2': 105, 'font': 'helvetica', 'size': 12, 'bold': 0, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0x88ff00, 'align': 'I', 'text': 'Lorem ipsum dolor sit amet, consectetur adipisici elit', 'priority': 2, 'multiline': 1}
    { 'name': 'box', 'type': 'B', 'x1': 15.0, 'y1': 15.0, 'x2': 185.0, 'y2': 260.0, 'font': 'helvetica', 'size': 0.0, 'bold': 0, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0, 'align': 'I', 'text': None, 'priority': 0, 'multiline': 0},
    { 'name': 'box_x', 'type': 'B', 'x1': 95.0, 'y1': 15.0, 'x2': 105.0, 'y2': 25.0, 'font': 'helvetica', 'size': 0.0, 'bold': 1, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0, 'align': 'I', 'text': None, 'priority': 2, 'multiline': 0},
    { 'name': 'line1', 'type': 'L', 'x1': 100.0, 'y1': 25.0, 'x2': 100.0, 'y2': 57.0, 'font': 'helvetica', 'size': 0, 'bold': 0, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0, 'align': 'I', 'text': None, 'priority': 3, 'multiline': 0},
    { 'name': 'barcode', 'type': 'BC', 'x1': 20.0, 'y1': 246.5, 'x2': 140.0, 'y2': 254.0, 'font': 'Interleaved 2of5 NT', 'size': 0.75, 'bold': 0, 'italic': 0,
'underline': 0, 'foreground': 0, 'background': 0, 'align': 'I', 'text': '2000000000010001590533801658120081', 'priority': 3, 'multiline': 0},
]

#here we instantiate the template
f = Template(format="A4", elements=elements,
             title="Sample Invoice")
f.add_page()

#we FILL some of the fields of the template with the information we want
#note we access the elements treating the template instance as a "dict"
f["company_name"] = "Sample Company"
f["company_logo"] = "docs/fpdf2-logo.png"

#and now we render the page
f.render("./template.pdf")
```

See template.py or [Web2Py] (Web2Py.md) for a complete example.

20.2.4 Example - Elements defined in CSV file

You define your elements in a CSV file "mycsvfile.csv" that will look like:

```
line0;L;20.0;12.0;190.0;12.0;times;0.5;0;0;0;16777215;C;;0;0;0.0
line1;L;20.0;36.0;190.0;36.0;times;0.5;0;0;0;16777215;C;;0;0;0.0
name0;T;21.0;14.0;104.0;25.0;times;16.0;0;0;0;0;16777215;L;name;2;0;0.0
title0;T;21.0;26.0;104.0;30.0;times;10.0;0;0;0;0;16777215;L;title;2;0;0.0
multiline;T;21.0;50.0;28.0;54.0;times;10.5;0;0;0;0;0xffff00;L;multi line;0;1;0.0
numeric_text;T;21.0;80.0;100.0;84.0;times;10.5;0;0;0;0;;R;007;0;0;0.0
empty_fields;T;21.0;100.0;100.0;104.0
rotated;T;21.0;80.0;100.0;84.0;times;10.5;0;0;0;0;;R;ROTATED;0;0;30.0
```

Remember that each line represents an element and each field represents one of the properties of the element in the following order: ('name','type','x1','y1','x2','y2','font','size','bold','italic','underline','foreground','background','align','text','priority', 'multiline', 'rotate') As noted above, most fields may be left empty, so a line is valid with only 6 items. The "empty_fields" line of the example demonstrates all that can be left away. In addition, for the barcode types "x2" may be empty.

Then you can use the file like this:

```
def test_template():
    f = Template(format="A4",
                 title="Sample Invoice")
    f.parse_csv("mycsvfile.csv", delimiter=";")
    f.add_page()
    f["name0"] = "Joe Doe"
    return f.render("./template.pdf")
```

21. Unicode

The FPDF class was modified adding UTF-8 support. Moreover, it embeds only the necessary parts of the fonts that are used in the document, making the file size much smaller than if the whole fonts were embedded. These features were originally developed for the mPDF project, and ported from Ian Back's sFPDF LGPL PHP version.

Before you can use UTF-8, you have to install at least one Unicode font in the font directory (or system font folder). Some free font packages are available for download (extract them into the font folder):

- DejaVu family: Sans, Sans Condensed, Serif, Serif Condensed, Sans Mono (Supports more than 200 languages)
- GNU FreeFont family: FreeSans, FreeSerif, FreeMono
- Indic (ttf-indic-fonts Debian and Ubuntu package) for Bengali, Devanagari, Gujarati, Gurmukhi (including the variants for Punjabi), Kannada, Malayalam, Oriya, Tamil, Telugu, Tibetan
- AR PL New Sung (firefly): The Open Source Chinese Font (also supports other east Asian languages)
- Alee (ttf-alee Arch Linux package): General purpose Hangul Truetype fonts that contain Korean syllable and Latin9 (iso8859-15) characters.
- Fonts-TLWG (formerly ThaiFonts-Scalable)

These fonts are included with this library's installers; see Free Font Pack for FPDF below for more information.

Then, to use a Unicode font in your script, pass `True` as the fourth parameter of `add_font`.

21.1 Example

This example uses several free fonts to display some Unicode strings. Be sure to install the fonts in the `font` directory first.

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

from fpdf import FPDF

pdf = FPDF()
pdf.add_page()

# Add a DejaVu Unicode font (uses UTF-8)
# Supports more than 200 languages. For a coverage status see:
# http://dejavu.svn.sourceforge.net/viewvc/dejavu/trunk/dejavu-fonts/langcover.txt
pdf.add_font('DejaVu', fname='DejaVuSansCondensed.ttf')
pdf.set_font('DejaVu', size=14)

text = u"""
English: Hello World
Greek: Γειά σου κόσμος
Polish: Witaj świecie
Portuguese: Olá mundo
Russian: Здравствуй, Мир
Vietnamese: Xin chào thế giới
Arabic: مرحبا العالم
Hebrew: שלום עולם
"""

for txt in text.split('\n'):
    pdf.write(8, txt)
    pdf.ln(8)

# Add a Indic Unicode font (uses UTF-8)
# Supports: Bengali, Devanagari, Gujarati,
#           Gurmukhi (including the variants for Punjabi)
#           Kannada, Malayalam, Oriya, Tamil, Telugu, Tibetan
pdf.add_font('gargi', fname='gargi.ttf')
pdf.set_font('gargi', size=14)
```

```
pdf.write(8, u'Hindi:              ')
pdf.ln(20)

# Add a AR PL New Sung Unicode font (uses UTF-8)
# The Open Source Chinese Font (also supports other east Asian languages)
pdf.add_font('fireflysung', fname='fireflysung.ttf')
pdf.set_font('fireflysung', size=14)
pdf.write(8, u'Chinese:     \n')
pdf.write(8, u'Japanese:        \n')
pdf.ln(10)

# Add a Alee Unicode font (uses UTF-8)
# General purpose Hangul truetype fonts that contain Korean syllable
# and Latin9 (iso8859-15) characters.
pdf.add_font('eunjin', fname='Eunjin.ttf')
pdf.set_font('eunjin', size=14)
pdf.write(8, u'Korean:      ')
pdf.ln(20)

# Add a Fonts-TLWG (formerly ThaiFonts-Scalable) (uses UTF-8)
pdf.add_font('waree', fname='Waree.ttf')
pdf.set_font('waree', size=14)
pdf.write(8, u'Thai:             ')
pdf.ln(20)

# Select a standard font (uses windows-1252)
pdf.set_font('helvetica', size=14)
pdf.ln(10)
pdf.write(5, 'This is standard built-in font')

pdf.output("unicode.pdf")
```

View the result here: unicode.pdf

21.2 Metric Files

FPDF will try to automatically generate metrics (i.e. character widths) about TTF font files to speed up their processing.

Such metrics are stored using the Python Pickle format (`.pkl` extension), by default in the font directory (ensure read and write permission!). Additional information about the caching mechanism is defined in the `add_font` method documentation.

TTF metric files often weigh about 650K, so keep that in mind if you use many TTF fonts and have disk size or memory limitations.

By design, metric files are not imported as they could cause a temporary memory leak if not managed properly (this could be an issue in a webserver environment with many processes or threads, so the current implementation discards metrics when FPDF objects are disposed).

In most circumstances, you will not notice any difference about storing metric files vs. generating them in each run on-the-fly (according basic tests, elapsed time is equivalent; YMMV).

Like the original PHP implementation, this library should work even if it could not store the metric file, and as no source code file is generated at runtime, it should work in restricted environments.

21.3 Free Font Pack and Copyright Restrictions

For your convenience, this library collected 96 TTF files in an optional "Free Unicode TrueType Font Pack for FPDF", with useful fonts commonly distributed with GNU/Linux operating systems (see above for a complete description). This pack is included in the Windows installers, or can be downloaded separately (for any operating system).

You could use any TTF font file as long embedding usage is allowed in the licence. If not, a runtime exception will be raised saying: "ERROR - Font file filename.ttf cannot be embedded due to copyright restrictions."

22. Emojis, Symbols & Dingbats

22.1 Emojis

Displaying emojis requires the use of a Unicode font file. Here is an example using the DejaVu font:

```
import fpdf

pdf = fpdf.FPDF()
pdf.add_font("DejaVuSans", fname="DejaVuSans.ttf")
pdf.set_font("DejaVuSans", size=64)
pdf.add_page()
pdf.multi_cell(0, txt="".join([chr(0x1F600 + x) for x in range(68)]))
pdf.set_font_size(32)
pdf.text(10, 270, "".join([chr(0x1F0A0 + x) for x in range(15)]))
pdf.output("fonts_emoji_glyph.pdf")
```

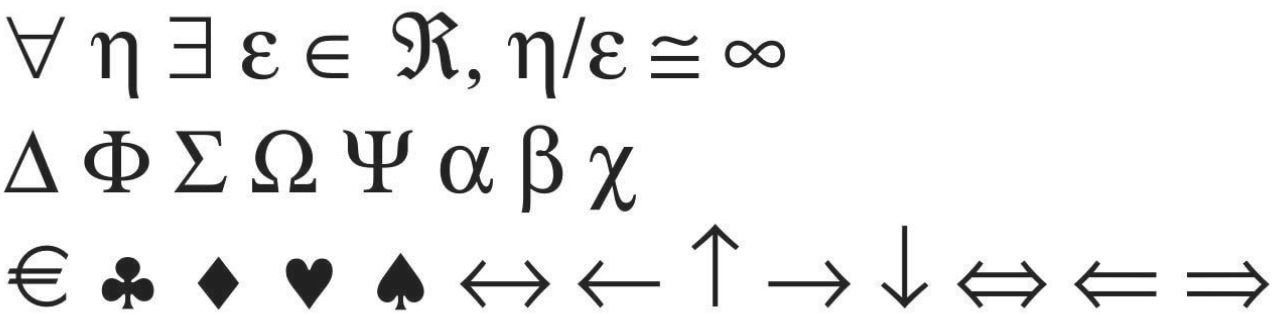
This code produces this PDF file: fonts_emoji_glyph.pdf

22.2 Symbols

The **Symbol** font is one of the built-in fonts in the PDF format. Hence you can include its symbols very easily:

```
import fpdf

pdf = fpdf.FPDF()
pdf.add_page()
pdf.set_font("symbol", size=36)
pdf.cell(h=16, txt="\u0022 \u0068 \u0024 \u0065 \u00ce \u00c2, \u0068/\u0065 \u0040 \u00a5", ln=1)
pdf.cell(h=16, txt="\u0044 \u0046 \u0053 \u0057 \u0059 \u0061 \u0062 \u0063", ln=1)
pdf.cell(h=16, txt="\u00a0 \u00a7 \u00a8 \u00a9 \u00aa \u00ab \u00ac \u00ad \u00ae \u00af \u00db \u00dc \u00de", ln=1)
pdf.output("symbol.pdf")
```

This results in:



The following table will help you find which characters map to which symbol: symbol.pdf. For reference, it was built using this script: symbol.py.

22.3 Dingbats

The **ZapfDingbats** font is one of the built-in fonts in the PDF format. Hence you can include its dingbats very easily:

```
import fpdf

pdf = fpdf.FPDF()
pdf.add_page()
pdf.set_font("zapfdingbats", size=36)
pdf.cell(txt="+ 3 8 A r \u00a6 } \u00a8 \u00a9 \u00aa \u00ab ~")
pdf.output("zapfdingbat.pdf")
```

This results in:

The following table will help you find which characters map to which dingbats: zapfdingbats.pdf. For reference, it was built using this script: zapfdingbats.py.

23. borb

Joris Schellekens made another excellent pure-Python library dedicated to reading & write PDF: borb. He even wrote a book about it, available publicly there: borb-examples.

23.1 Creating a document with `fpdf2` and transforming it into a `borb.pdf.document.Document`

```
from io import BytesIO
from borb.pdf.pdf import PDF
from fpdf import FPDF

pdf = FPDF()
pdf.set_title('Initiating a borb doc from a FPDF instance')
pdf.set_font('helvetica', size=12)
pdf.add_page()
pdf.cell(txt="Hello world!")

doc = PDF.loads(BytesIO(pdf.output()))
print(doc.get_document_info().get_title())
```

24. Usage in web API

24.1 Django

Usage in a view:

```
from fpdf import FPDF
from django.http import HttpResponse


def report(request):
    pdf = FPDF()
    ...
    return HttpResponse(bytes(pdf.output()), content_type='application/pdf')
```

24.2 web2py

Usage of the original PyFPDF lib with web2py is described here: https://github.com/reingart/pyfpdf/blob/master/docs/Web2Py.md

PyFPDF is included in `web2py` since release `1.85.2`: https://github.com/web2py/web2py/tree/master/gluon/contrib/fpdf

25. Development

This page has summary information about developing the PyPDF library.

25.1 History

This project, `fpdf2` is a *fork* of the `PyFPDF` project, which can be found on GitHub at reingart/pyfpdf.

Its aim is to keep the library up to date, to fulfill the goals of its original roadmap and provide a general overhaul of the codebase to address technical debt keeping features from being added and bugs to be eradicated.

More on `PyFPDF`:

This project started as Python fork of the FPDF PHP library. Later, code for native reading TTF fonts was added. FPDF has not been updated since 2011. See also the TCPDF library.

Until 2015 the code was developed at Google Code. Now the main repository is at Github.

You can also view the old repository, old issues, and old wiki.

25.2 Usage

- PyPI download stats - Downloads per release on Pepy
- packages using `fpdf2` can be listed using GitHub Dependency graph: Dependents, Wheelodex or Watchman Pypi. Some are also listed on its libraries.io page.

25.3 Repository structure

- `[.github]` - GitHub Actions configuration
- `[docs]` - documentation folder
- `[fpdf]` - library source
- `[scripts]` - utilities to validate PDF files & publish the package on Pypi

- `[test]` - non-regression tests
- `[tutorial]` - tutorials (see also Tutorial)
- `README.md` - Github and PyPI ReadMe
- `CHANGELOG.md` - details of each release content
- `LICENSE` - code license information
- `CODEOWNERS` - define individuals or teams responsible for code in this repository
- `CONTRIBUTORS.md` - the people who helped build this library ❤️
- `setup.cfg`, `setup.py`, `MANIFEST.in` - packaging configuration to publish a package on Pypi
- `mkdocs.yml` - configuration for MkDocs
- `tox.ini` - configuration for Tox
- `.pylintrc` - configuration for Pylint

25.4 Code auto-formatting

We use black as a code prettifier. This "*uncomprimising Python code formatter*" must be installed in your development environment (`pip install black`) in order to auto-format source code before any commit.

25.5 Linting

We use pylint as a static code analyzer to detect potential issues in the code.

In case of special "false positive" cases, checks can be disabled locally with `#pylint disable=XXX` code comments, or globally through the `.pylintrc` file.

25.6 Pre-commit hook

If you use a UNIX system, you can place the following shell code in `.git/hooks/pre-commit` in order to always invoke `black` & `pylint` before every commit:

```
#!/bin/bash
git_cached_names() { git diff --cached --name-only --diff-filter=ACM; }
if grep -IRF generate=True $(git_cached_names | grep 'test.*\.py$'); then
    echo '`generate=True` left remaining in a call to assert_pdf_equal'
    exit 1
fi
modified_py_files=$(git_cached_names | grep '\.py$')
modified_fpdf_files=$(git_cached_names | grep '^fpdf.*\.py$')
# If any Python files were modified, format them:
if [ -n "$modified_py_files" ]; then
    if ! black --check $modified_py_files; then
        black $modified_py_files
        exit 1
    fi
    # If fpdf/ files were modified, lint them:
    [[ $modified_fpdf_files == "" ]] || pylint $modified_fpdf_files
fi
```

It will abort the commit if `pylint` found issues or `black` detect non-properly formatted code. In the later case though, it will auto-format your code and you will just have to run `git commit -a` again.

25.7 Testing

25.7.1 Running tests

To run tests, `cd` into `fpdf2` repository, install the dependencies using `pip install -r test/requirements.txt`, and run `pytest`.

You can run a single test by executing: `pytest -k function_name`.

Alternatively, you can use Tox. It is self-documented in the `tox.ini` file in the repository. To run tests for all versions of Python, simply run `tox`. If you do not want to run tests for all versions of python, run `tox -e py39` (or your version of Python).

25.7.2 Why is a test failing?

If there are some failing tests after you made a code change, it is usually because **there are difference between an expected PDF generated and the actual one produced**.

Calling `pytest -vv` will display **the difference of PDF source code** between the expected & actual files, but that may be difficult to understand,

You can also have a look at the PDF files involved by navigating to the temporary test directory that is printed out during the test failure:

```
================================== FAILURES ===================================
_____________________________ test_html_simple_table _____________________________

tmp_path = PosixPath('/tmp/pytest-of-runner/pytest-0/test_html_simple_table0')
```

This directory contains the **actual** & **expected** files, that you can vsualize to spot differences:

```
$ ls /tmp/pytest-of-runner/pytest-0/test_html_simple_table0
actual.pdf
actual_qpdf.pdf
expected_qpdf.pdf
```

25.7.3 assert_pdf_equal & writing new tests

When a unit test generates a PDF, it is recommended to use the `assert_pdf_equal` utility function in order to validate the output. It relies on the very handy qpdf CLI program to generate a PDF that is easy to compare: annotated, strictly formatted, with uncompressed internal streams. You will need to have its binary in your `$PATH`, otherwise `assert_pdf_equal` will fall back to hash-based comparison.

All generated PDF files (including those processed by `qpdf`) will be stored in `/tmp/pytest-of-USERNAME/pytest-current/NAME_OF_TEST/`. By default, three last test runs will be saved and then automatically deleted, so you can check the output in case of a failed test.

In order to generate a "reference" PDF file, simply call `assert_pdf_equal` once with `generate=True`.

25.8 GitHub pipeline

A GitHub Actions pipeline is executed on every commit on the `master` branch, and for every *Pull Request*.

It performs all validation steps detailed above: code checking with `black`, static code analysis with `pylint`, unit tests... *Pull Requests* submitted must pass all those checks in order to be approved. Ask maintainers through comments if some errors in the pipeline seem obscure to you.

25.8.1 Release checklist

1. complete `CHANGELOG.md` and add the version & date of the new release
2. bump `FPDF_VERSION` in `fpdf/fpdf.py`
3. `git commit` & `git push`
4. check that the GitHub Actions succeed, and that a new release appears on Pypi
5. perform a GitHub release, taking the description from the `CHANGELOG.md`. It will create a new `git` tag.
6. Announce the release on r/pythonnews

25.9 Documentation

The standalone documentation is in the `docs` subfolder, written in Markdown. Building instructions are contained in the configuration file `mkdocs.yml` and also in `.github/workflows/continuous-integration-workflow.yml`.

Additional documentation is generated from inline comments, and is available in the project home page.

After being committed to the master branch, code documentation is automatically uploaded to GitHub Pages.

There is a useful one-page example Python module with docstrings illustrating how to document code: pdoc3 example_pkg.

To preview the Markdown documentation, launch a local rendering server with:

```
mkdocs serve
```

To preview the API documentation, launch a local rendering server with:

```
pdoc --html -o public/ fpdf --http :
```

25.10 PDF spec & new features

The **PDF 1.7 spec** is available on Adobe website: PDF32000_2008.pdf.

It may be intimidating at first, but while technical, it is usually quite clear and understandable.

It is also a great place to look for new features for `fpdf2`: there are still many PDF features that this library does not support.

26. FAQ

See Project Home for an overall introduction.

26.1 What is fpdf2?

`fpdf2` is a library with low-level primitives to easily generate PDF documents.

This is similar to ReportLab's graphics canvas, but with some methods to output "fluid" cells ("flowables" that can span multiple rows, pages, tables, columns, etc).

It has methods ("hooks") that can be implemented in a subclass: `headers` and `footers`.

Originally developed in PHP several years ago (as a free alternative to proprietary C libraries), it has been ported to many programming languages, including ASP, C++, Java, Pl/SQL, Ruby, Visual Basic, and of course, Python.

For more information see: http://www.fpdf.org/en/links.php

26.2 What is this library not?

This library is not a:

- charts or widgets library. But you can import PNG or JPG images, use PIL or any other library, or draw the figures yourself.
- "flexible page layout engine" like Reportlab PLATYPUS. But it can do columns, chapters, etc.; see the Tutorial.
- XML or object definition language like Geraldo Reports, Jasper Reports, or similar. But look at write_html for simple HTML reports and Templates for fill-in-the-blank documents.
- PDF text extractor, converter, splitter or similar.

26.3 How does this library compare to ...?

The API is geared toward giving the user access to features of the Portable Document Format as they are described in the Adobe PDF Reference Manual, this bypasses needless complexities for simpler use cases.

It is small:

```
$ du -sh fpdf
272K    fpdf

$ scc fpdf
─────────────────────────────────────────────────────────────────────────────
Language                 Files     Lines   Blanks  Comments     Code Complexity
─────────────────────────────────────────────────────────────────────────────
Python                      14      8204      142       232     7830        192
```

It includes `cell` and `multi_cell` primitives to draw fluid document like invoices, listings and reports, and includes basic support for HTML rendering.

Compared to other solutions, this library should be easier to use and adapt for most common documents (no need to use a page layout engine, style sheets, templates, or stories...), with full control over the generated PDF document (including advanced features and extensions).

Check also the list of features on the home page.

26.4 What does the code look like?

Following is an example similar to the Reportlab one in the book of web2py. Note the simplified import and usage: (http://www.web2py.com/book/default/chapter/09?search=pdf#ReportLab-and-PDF)

```
from fpdf import FPDF

def get_me_a_pdf():
    title = "This The Doc Title"
    heading = "First Paragraph"
    text = 'bla ' * 10000

    pdf = FPDF()
    pdf.add_page()
    pdf.set_font('Times', 'B', 15)
    pdf.cell(w=210, h=9, txt=title, border=0, ln=1, align='C', fill=False)
    pdf.set_font('Times', 'B', 15)
    pdf.cell(w=0, h=6, txt=heading, border=0, ln=1, align='L', fill=False)
    pdf.set_font('Times', '', 12)
    pdf.multi_cell(w=0, h=5, txt=text)
    response.headers['Content-Type'] = 'application/pdf'
    return pdf.output()
```

With Reportlab:

```
from reportlab.platypus import *
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.rl_config import defaultPageSize
from reportlab.lib.units import inch, mm
from reportlab.lib.enums import TA_LEFT, TA_RIGHT, TA_CENTER, TA_JUSTIFY
from reportlab.lib import colors
from uuid import uuid4
from cgi import escape
import os

def get_me_a_pdf():
    title = "This The Doc Title"
    heading = "First Paragraph"
    text = 'bla ' * 10000

    styles = getSampleStyleSheet()
    tmpfilename = os.path.join(request.folder, 'private', str(uuid4()))
    doc = SimpleDocTemplate(tmpfilename)
    story = []
    story.append(Paragraph(escape(title), styles["Title"]))
    story.append(Paragraph(escape(heading), styles["Heading2"]))
    story.append(Paragraph(escape(text), styles["Normal"]))
    story.append(Spacer(1, 2 * inch))
    doc.build(story)
    data = open(tmpfilename, "rb").read()
    os.unlink(tmpfilename)
    response.headers['Content-Type'] = 'application/pdf'
    return data
```

26.5 Does this library have any framework integration?

Yes, if you use web2py, you can make simple HTML reports that can be viewed in a browser, or downloaded as PDF.

Also, using web2py DAL, you can easily set up a templating engine for PDF documents.

Look at Web2Py for examples.

26.6 What is the development status of this library?

This library was improved over the years since the initial port from PHP. As of 2021, it is **stable** and actively maintained, with bug fixes and new features developped regularly.

In contrast, `write_html` support is not complete, so it must be considered in beta state.

26.7 What is the license of this library (fpdf2)?

LGPL v3.0.

Original FPDF uses a permissive license: http://www.fpdf.org/en/FAQ.php#q1

"FPDF is released under a permissive license: there is no usage restriction. You may embed it freely in your application (commercial or not), with or without modifications."

FPDF version 1.6's license.txt says: http://www.fpdf.org/es/dl.php?v=16&f=zip

Permission is hereby granted, free of charge, to any person obtaining a copy of this software to use, copy, modify, distribute, sublicense, and/or sell copies of the software, and to permit persons to whom the software is furnished to do so.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.

The fpdf.py library is a revision of a port by Max Pat. The original source uses the same licence: http://www.fpdf.org/dl.php?id=94

```
# * Software: FPDF
# * Version:  1.53
# * Date:     2004-12-31
# * Author:   Olivier PLATHEY
# * License:  Freeware
# *
# * You may use and modify this software as you wish.
# * Ported to Python 2.4 by Max (maxpat78@yahoo.it) on 2006-05
```

To avoid ambiguity (and to be compatible with other free software, open source licenses), LGPL was chosen for the Google Code project (as freeware isn't listed).

Some FPDF ports had chosen similar licences (wxWindows Licence for C++ port, MIT licence for Java port, etc.): http://www.fpdf.org/en/links.php

Other FPDF derivatives also choose LGPL, such as sFPDF by Ian Back.