

fpdf2 manual

A minimalist PDF creation library for Python

Table of contents

1. fpdf2	4
1.1 Main features	4
1.2 Tutorials	5
1.3 Installation	6
1.4 Community	6
1.5 Misc	8
2. Tutorial	9
2.1 Hello World with fpdf2	9
2.2 Tutorial	10
3. Page Layout	16
3.1 Page format and orientation	16
3.2 Margins	18
3.3 Introduction	19
3.4 How to use Templates	19
3.5 Details - Template definition	21
3.6 How to create a template	23
3.7 Example - Python dict	23
3.8 Example - Elements defined in JSON file	24
3.9 Example - Elements defined in CSV file	24
3.10 Text Flow Regions	25
3.11 Tables	32
4. Text Content	45
4.1 Adding Text	45
4.2 Line breaks	49
4.3 Page breaks	50
4.4 Text styling	51
4.5 Fonts and Unicode	56
4.6 Text Shaping	61
4.7 Bidirectional Text	63
4.8 Emojis, Symbols & Dingbats	65
4.9 HTML	67
5. Graphics Content	70
5.1 Images	70
5.2 Shapes	74
5.3 Transformations	86

5.4	Transparency	89
5.5	Barcodes	91
5.6	Drawing	96
5.7	Scalable Vector Graphics (SVG)	101
5.8	Charts & graphs	106
6.	PDF Features	113
6.1	Links	113
6.2	Metadata	115
6.3	Annotations	116
6.4	Presentations	119
6.5	Document outline & table of contents	120
6.6	Encryption	122
6.7	Signing	124
6.8	File attachments	125
7.	Mixing other libs	126
7.1	borb	126
7.2	Combine with livereload	127
7.3	Combine with mistletoe to use Markdown	128
7.4	Combine with pypdf	130
7.5	Combine with pdfwr	132
7.6	Matplotlib, Pandas, Plotly, Pygal	134
7.7	Templating with Jinja	135
7.8	Usage in web APIs	136
7.9	Database storage	139
8.	Development	140
8.1	Development	140
8.2	Logging	146
9.	History	148
9.1	How fpdf2 came to be	148
9.2	Compatibility between PyFPDF & fpdf2	148
10.	FAQ	150
10.1	What is fpdf2?	150
10.2	What is this library not ?	150
10.3	How does this library compare to ...?	150
10.4	What does the code look like?	151
10.5	Does this library have any framework integration?	151
10.6	What is the development status of this library?	152
10.7	What is the license of this library (fpdf2)?	152

1. fpdf2

`fpdf2` is a library for simple & fast PDF document generation in Python. It is a fork and the successor of `PyFPDF` (*cf.* [history](#)).

Latest Released Version: `pypi` `v2.7.9`



```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font('helvetica', size=12)
pdf.cell(text="hello world")
pdf.output("hello_world.pdf")
```

Go try it **now** online in a Jupyter notebook:



Open in Colab

or



Open In nbviewer

1.1 Main features

- Easy to use, with a user-friendly [API](#), and easy to extend
- Python 3.7+ 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)
- Internal / external [links](#)

- Embedding images, including transparency and alpha channel, using [Pillow \(Python Imaging Library\)](#)
- Arbitrary path drawing and basic [SVG](#) import
- Embedding [barcodes](#), [charts & graphs](#), [emojis](#), [symbols & dingbats](#)
- [Tables](#), and also [cell / multi-cell / plaintext writing](#), with [automatic page breaks](#), line break and text justification
- Choice of measurement unit, page format & margins. Optional page header and footer
- Basic [conversion from HTML to PDF](#)
- A [templating system](#) to render PDFs in batches
- Images & links alternative descriptions, for accessibility
- Table of contents & [document outline](#)
- [Document encryption](#) & [document signing](#)
- [Annotations](#), including text highlights, and [file attachments](#)
- [Presentation mode](#) with control over page display duration & transitions
- Optional basic Markdown-like styling: `**bold**`, `__italics__`
- It has very few dependencies: [Pillow](#), [defusedxml](#), & [fonttools](#)
- Can render [mathematical equations & charts](#)
- Many example scripts available throughout this documentation, including usage examples with [Django](#), [Flask](#), [FastAPI](#), [streamlit](#), AWS lambdas... : [Usage in web APIs](#)
- more than 1300 unit tests with `qpdf`-based PDF diffing, and PDF samples validation using 3 different checkers:



1.2 Tutorials

- [English](#)
- [Deutsch](#)
- [Italian](#)

- [español](#)
- [français](#)
-
- [português](#)
- [Русский](#)
- [Ελληνικά](#)
- [עברית](#)
- [简体中文](#)
-
-
- [日本語](#)
- [Dutch](#)
- [Polski](#)

1.3 Installation

From [PyPI](#):

```
pip install fpdf2
```

To get the latest, unreleased, development version straight from the development branch of this repository:

```
pip install git+https://github.com/py-pdf/fpdf2.git@master
```

Development: check the [dedicated documentation page](#).

1.3.1 Displaying deprecation warnings

`DeprecationWarning`s are not displayed by Python by default.

Hence, every time you use a newer version of `fpdf2`, we strongly encourage you to execute your scripts with the `-Wd` option (*cf.* [documentation](#)) in order to get warned about deprecated features used in your code.

This can also be enabled programmatically with `warnings.simplefilter('default', DeprecationWarning)`.

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

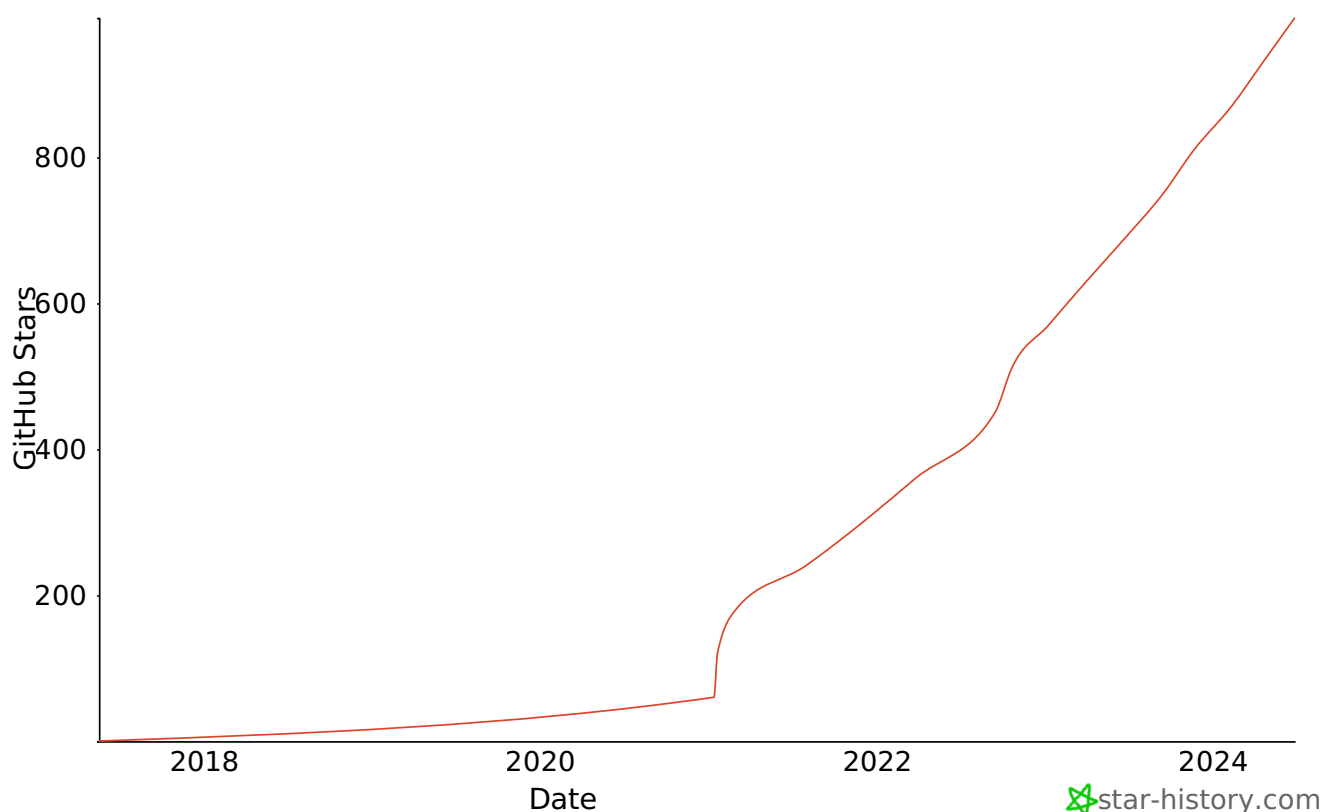
- **Harvard University** uses `fpdf2` in their [CS50 introductory class](#)
- **Undying Dusk** : a **video game in PDF format**, with a gameplay based on exploration and logic puzzles, in the tradition of dungeon crawlers
- **OpenDroneMap** : a command line toolkit for processing aerial drone imagery
- **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
- [wudududu/extract-video-ppt](#) : create a one-page-per-frame PDF from a video or PPT file. `fpdf2` also has a demo script to convert a GIF into a one-page-per-frame PDF: [gif2pdf.py](#)
- [csv2pdf](#) : convert CSV files to PDF files easily
- [Planet-Matriarchy-RPG-CharGen](#) : a PyQt based desktop application (= `.exe` under Windows) that provides a RPG character sheet generator

1.4.3 Usage statistics

- [PyPI download stats](#) - Downloads per release on [Pepy](#)
- [pip trends: fpdf2 VS other PDF rendering libs](#)
- packages using `fpdf2` can be listed using [GitHub Dependency graph: Dependents](#), [Wheelodex](#) or [Watchman Pypi](#). Some are also listed on [its libraries.io page](#).

Star History



1.4.4 Related

- Looking for alternative libraries? Check out [pypdf](#), [borb](#), [pikepdf](#), [WeasyPrint](#), [pydyf](#) and [PyMuPDF](#): [features comparison](#), [examples](#), [Jupyter notebooks](#). We have some documentations about combining `fpdf2` with [borb](#) & [pypdf](#).
- [Create PDFs with Python](#) : a series of tutorial videos by bvalgard
- [GitHub gist providing borders around any fpdf2 area](#), by [@hyperstown](#)
- [digidigital/Extensions-and-Scripts-for-pyFPDF-fpdf2](#) : scripts ported from PHP to add transparency 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...

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

 [Open in Colab](#)

 [Open In nbviewer](#)

2.1 Hello World with fpdf2

This [Jupyter notebook](#) demonstrates some basic usage of the Python [fpdf2](#) library

```
# Installation of fpdf2 with PIP:
!pip install fpdf2
```

```
# Enable deprecation warnings:
import warnings
warnings.simplefilter('default', DeprecationWarning)
```

```
# Generate a PDF:
from fpdf import FPDF
pdf = FPDF()
pdf.add_page()
pdf.set_font('helvetica', size=48)
pdf.cell(text="hello world")
pdf_bytes = pdf.output()
```

```
# Display the PDF in the notebook by embedding it as HTML content:
WIDTH, HEIGHT = 800, 400
from base64 import b64encode
from IPython.display import display, HTML
base64_pdf = b64encode(pdf_bytes).decode("utf-8")
display(HTML(f'<embed height="{HEIGHT}" src="data:application/pdf;base64,{base64_pdf}" type="application/pdf" width="{WIDTH}" />'))
```

```
# Display a download button:
display(HTML(f'<a download="fpdf2-demo.pdf" href="data:application/pdf;base64,{base64_pdf}">Click to download PDF</a>'))
```

[Click to download PDF](#)

To continue learning about `fpdf2`, check our tutorial: - [English](#) - [Deutsch](#) - [español](#) - [português](#) - [Русский](#) - [Italian](#) - [français](#) - [Ελληνικά](#) - [עברית](#) - [Dutch](#) - [Polski](#)

2.2 Tutorial

Methods full documentation: [fpdf.FPDF](#) [API doc](#)

- [Tutorial](#)
- [Tuto 1 - Minimal Example](#)
- [Tuto 2 - Header, footer, page break and image](#)
- [Tuto 3 - Line breaks and colors](#)
- [Tuto 4 - Multi Columns](#)
- [Tuto 5 - Creating Tables](#)
- [Tuto 6 - Creating links and mixing text styles](#)

2.2.1 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.', new_x="LMARGIN", new_y="NEXT", 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.

2.2.2 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", border=1, align="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}", align="C")

# Instantiation of inherited class
pdf = PDF()
pdf.add_page()
pdf.set_font("Times", size=12)
for i in range(1, 41):
    pdf.cell(0, 10, f"Printing line number {i}", new_x="LMARGIN", new_y="NEXT")
pdf.output("new-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 (this special value can be changed by `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`.

2.2.3 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,
    border=1,
    new_x="LMARGIN",
    new_y="NEXT",
    align="C",
    fill=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()}", align="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}",
        new_x="LMARGIN",
        new_y="NEXT",
        align="L",
        fill=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.

2.2.4 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 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,
            border=1,
            new_x="LMARGIN",
            new_y="NEXT",
            align="C",
            fill=True,
        )
        self.ln(10)

    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()}", align="C")

    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}",
            new_x="LMARGIN",
            new_y="NEXT",
            border="L",
            fill=True,
        )
        self.ln(4)

    def chapter_body(self, fname):
        # Reading text file:
        with open(fname, "rb") as fh:
            txt = fh.read().decode("latin-1")
        with self.text_columns(
            ncols=3, gutter=5, text_align="J", line_height=1.19
        ) as cols:
            # Setting font: Times 12
            self.set_font("Times", size=12)
            cols.write(txt)
            cols.ln()
            # Final mention in italics:
            self.set_font(style="I")
            cols.write("(end of excerpt)")

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

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 `text_columns` method. It collects all the text, possibly in increments, and distributes it across the requested number of columns, automatically inserting page breaks as necessary. Note that while the `TextColumns` instance is active as a context manager, text styles and other font properties can be changed. Those changes will be contained to the context. Once it is closed the previous settings will be reinstated.

2.2.5 Tuto 5 - Creating Tables

This tutorial will explain how to create two different tables, to demonstrate what can be achieved with some simple adjustments.

```
import csv
from fpdf import FPDF
from fpdf.fonts import FontFace
from fpdf.enums import TableCellFillMode

with open("countries.txt", encoding="utf8") as csv_file:
    data = list(csv.reader(csv_file, delimiter=","))

pdf = FPDF()
pdf.set_font("helvetica", size=14)

# Basic table:
pdf.add_page()
with pdf.table() as table:
    for data_row in data:
        row = table.row()
        for datum in data_row:
            row.cell(datum)

# Styled table:
pdf.add_page()
pdf.set_draw_color(255, 0, 0)
pdf.set_line_width(0.3)
headings_style = FontFace(emphasis="BOLD", color=255, fill_color=(255, 100, 0))
with pdf.table(
    borders_layout="NO_HORIZONTAL_LINES",
    cell_fill_color=(224, 235, 255),
    cell_fill_mode=TableCellFillMode.ROWS,
    col_widths=(42, 39, 35, 42),
    headings_style=headings_style,
    line_height=6,
    text_align=("LEFT", "CENTER", "RIGHT", "RIGHT"),
    width=160,
) as table:
    for data_row in data:
        row = table.row()
        for datum in data_row:
            row.cell(datum)

pdf.output("tuto5.pdf")
```

Resulting PDF - Countries CSV data

The first example is achieved in the most basic way possible, feeding data to `FPDF.table()`. The result is rudimentary but very quick to obtain.

The second table brings some improvements: colors, limited table width, reduced line height, centered titles, columns with custom widths, figures right aligned... Moreover, horizontal lines have been removed. This was done by picking a `borders_layout` among the available values: `TableBordersLayout`.

2.2.6 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

pdf = FPDF()

# 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(page=2)
pdf.write(5, "here", link)
pdf.set_font()
```

```
# Second page:
pdf.add_page()
pdf.image(
    "../docs/fpdf2-logo.png", 10, 10, 50, 0, "", "https://py-pdf.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://py-pdf.github.io/fpdf2/">https://py-pdf.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 page within the document.

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. Page Layout

3.1 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.

3.1.1 Per-page format, orientation and background

`.set_page_background()` lets you set a background for all pages following this call until the background is removed. The value must be of type `str`, `io.BytesIO`, `PIL.Image.Image`, `drawing.DeviceRGB`, `tuple` or `None`.

The following code snippet illustrates how to configure different page formats for specific pages as well as setting different backgrounds and then removing it:

```
from fpdf import FPDF

pdf = FPDF()
pdf.set_font("Helvetica")
pdf.set_page_background((252,212,255))
for i in range(9):
    if i == 6:
        pdf.set_page_background('image_path.png')
        pdf.add_page(format=(210 * (1 - i/10), 297 * (1 - i/10)))
        pdf.cell(text=str(i))
    pdf.set_page_background(None)
pdf.add_page(same=True)
pdf.cell(text="9")
pdf.output("varying_format.pdf")
```

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

3.1.2 Page layout & zoom level

`set_display_mode()` allows to set the **zoom level**: pages can be displayed entirely on screen, occupy the full width of the window, use the real size, be scaled by a specific zooming factor or use the viewer default (configured in its *Preferences* menu).

The **page layout** can also be specified: single page at a time, continuous display, two columns or viewer default.

```
from fpdf import FPDF

pdf = FPDF()
pdf.set_display_mode(zoom="default", layout="TWO_COLUMN_LEFT")
pdf.set_font("helvetica", size=30)
pdf.add_page()
pdf.cell(text="page 1")
pdf.add_page()
pdf.cell(text="page 2")
pdf.output("two-column.pdf")
```

3.1.3 Viewer preferences

```
from fpdf import FPDF, ViewerPreferences

pdf = FPDF()
pdf.viewer_preferences = ViewerPreferences(
    hide_toolbar=True,
    hide_menubar=True,
    hide_window_u_i=True,
    fit_window=True,
    center_window=True,
    display_doc_title=True,
    non_full_screen_page_mode="USE_OUTLINES",
```



```
)  
pdf.set_font("helvetica", size=30)  
pdf.add_page()  
pdf.cell(text="page 1")  
pdf.add_page()  
pdf.cell(text="page 2")  
pdf.output("viewer-prefs.pdf")
```

3.1.4 Full screen

```
from fpdf import FPDF  
  
pdf = FPDF()  
pdf.page_mode = "FULL_SCREEN"  
pdf.output("full-screen.pdf")
```

3.2 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](#)

3.3 Introduction

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, a JSON 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.

3.4 How to use Templates

There are two approaches to using templates.

3.4.1 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 [py-pdf.github.io: class Template](https://py-pdf.github.io/class%20Template).

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"
```

3.4.2 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 [py-pdf.github.io: class FlexTemplate](https://py-pdf.github.io/class_FlexTemplate).

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

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

3.5 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, name/value pairs in a JSON object, 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 dict/JSON, enabled with True/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)
- in JSON, a decimal value or a HTML style "#RRGGBB" string (6 digits) can be given.
- *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/JSON, None/null for single line, True/true for multicells (multiple lines), False/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
- **wrapmode**: optionally set wrapmode to "CHAR" to support multiline line wrapping on characters instead of words
- *optional*
- default: 'WORD'

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).

3.6 How to create a template

A template can be created in several ways:

- By defining everything directly as a Python dictionary
- By reading the template definition from a JSON document with `.parse_json()`
- By reading the template definition from a CSV document with `.parse_csv()`
- By defining the template in a database (this applies to [Web2Py] (Web2Py.md) integration)

3.7 Example - Python dict

```
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,
      'align': 'C', 'text': 'logo', 'priority': 2, 'multiline': False},
    { '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, 'align': 'C', 'text': '', 'priority': 2, 'multiline': False},
    { 'name': 'multiline_text', 'type': 'T', 'x1': 20, 'y1': 100, 'x2': 40, 'y2': 105, 'font': 'helvetica', 'size': 12, 'bold': 0, 'italic': 0, 'underline': 0,
      'background': '0x88ff00', 'align': 'C', 'text': 'Lorem ipsum dolor sit amet, consectetur adipisicing elit', 'priority': 2, 'multiline': True, 'wrapmode': 'WORD'},
    { '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,
      'align': 'C', 'text': None, 'priority': 0, 'multiline': False},
    { '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,
      'align': 'C', 'text': None, 'priority': 2, 'multiline': False},
    { '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,
      'align': 'C', 'text': None, 'priority': 3, 'multiline': False},
    { '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, 'align': 'C', 'text': '20000000001000159053338016581200810081', 'priority': 3, 'multiline': False},
]

#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.

3.8 Example - Elements defined in JSON file

New in  2.7.10

The JSON file must consist of an array of objects. Each object with its name/value pairs define a template element:

```
[
  {
    "name": "employee_name",
    "type": "T",
    "x1": 20,
    "y1": 75,
    "x2": 118,
    "y2": 90,
    "font": "helvetica",
    "size": 12,
    "bold": true,
    "underline": true,
    "text": ""
  }
]
```

Then you import and use that template as follows:

```
def test_template():
    f = Template(format="A4", title="Template Demo")
    f.parse_json("myjsonfile.json")
    f.add_page()
    f["employee_name"] = "Joe Doe"
    return f.render("./template.pdf")
```

3.9 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;16777215;L;name;2;0;0.0
title0;T;21.0;26.0;104.0;30.0;times;10.0;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;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;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;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', 'wrapmode') 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")
```


3.10 Text Flow Regions

New in  2.7.6

3.10.1 Text Flow Regions

Notice: As of fpdf2 release 2.7.6, this is an experimental feature. Both the API and the functionality may change before it is finalized, without prior notice.

Text regions are a hierarchy of classes that enable to flow text within a given outline. In the simplest case, it is just the running text column of a page. But it can also be a sequence of outlines, such as several parallel columns or the cells of a table. Other outlines may be combined by addition or subtraction to create more complex shapes.

There are two general categories of regions. One defines boundaries for running text that will just continue in the same manner one the next page. Those include columns and tables. The second category are distinct shapes. Examples would be a circle, a rectangle, a polygon of individual shape or even an image. They may be used individually, in combination, or to modify the outline of a multipage column. Shape regions will typically not cause a page break when they are full. In the future, a possibility to chain them may be implemented, so that a new shape will continue with the text that didn't fit into the previous one.

The currently implemented text regions are: * [Text Columns](#)

Other types like Table cells, shaped regions and combinations are still in the design phase, see [Quo vadis, .write\(\)?](#).

General Operation

Using the different region types and combination always follows the same pattern. The main difference to the normal `FPDF.write()` method is that all added text will first be buffered, and only gets rendered on the page when the context of the region is closed. This is necessary so that text can be aligned within the given boundaries even if its font, style, or size are arbitrarily varied along the way.

- Create the region instance with an `FPDF` method, , for example `text_columns()`.
- Use the `.write()` method of this text region in order to feed text into its buffer.
- Best practise is to use the region instance as a context manager for filling.
- Text will be rendered automatically after closing the context.
- When used as a context manager, you can change all text styling parameters within that context, and they will be used by the added text, but won't leak to the surroundings
- Alternatively, eg. for filling a single column of text with the already existing settings, just use the region instance as is. In that case, you'll have to explicitly use the `render()` method after adding the text.
- Within a region, paragraphs can be inserted. The primary purpose of a paragraph is to apply a different horizontal alignment than the surrounding text. It is also possible to apply margins to the top and bottom of each paragraph.



The graphic shows the relationship of page, text areas and paragraphs (with varying alignment) for the example of a two-column layout.

TEXT START POSITION

When rendering, the vertical start position of the text will be at the lowest one out of: * the current y position * the top of the region (if it has a defined top) * the top margin of the page.

The horizontal start position will be either at the current x position, if that lies within the boundaries of the region/column, or at the left edge of the region. In both horizontal and vertical positioning, regions with multiple columns may follow additional rules and restrictions.

INTERACTION BETWEEN REGIONS

Several region instances can exist at the same time. But only one of them can act as context manager at any given time. It is not currently possible to activate them recursively. But it is possible to use them intermittingly. This will probably most often make sense between a columnar region and a table or a graphic. You may have some running text ending at a given height, then insert a table/graphic, and finally continue the running text at the new height below the table within the existing column(s).

COMMON PARAMETERS

All types of text regions have the following constructor parameters in common:

- `text` (str, optional) - text content to add to the region. This is a convenience parameter for cases when all text is available in one piece, and no partition into paragraphs (possibly with different parameters) is required. (Default: None)
- `text_align` (Align/str, optional) - the horizontal alignment of the text in the region. (Default: Align.L)
- `line_height` (float, optional) - This is a factor by which the line spacing will be different from the font height. It works similar to the attribute of the same name in HTML/CSS. (default: 1.0)
- `print_sh` (bool, optional) - Treat a soft-hyphen (`\u00ad`) as a printable character, instead of a line breaking opportunity. (Default: False)
- `skip_leading_spaces` (default: False) - This flag is primarily used by `write_html()`, but may also have other uses. It removes all space characters at the beginning of each line.
- `wrapmode` (default "WORD") -
- `image` (str or PIL.Image.Image or io.BytesIO, optional) - An image to add to the region. This is a convenience parameter for cases when no further text or images need to be added to the paragraph. If both "text" and "image" arguments are present, the text will be inserted first. (Default: None)
- `image_fill_width` (bool, optional) - Indicates whether to increase the size of the image to fill the width of the column. Larger images will always be reduced to column width. (Default: False)

All of those values can be overridden for each individual paragraph.

COMMON METHODS

- `.paragraph()` [see characteristic parameters below] - establish a new paragraph in the text. The text added to this paragraph will start on a new line.
- `.write(text: str, link: = None)` - write text to the region. This is only permitted when no explicit paragraph is currently active.
- `.image()` [see characteristic parameters below] - insert a vector or raster image in the region, flowing with the text like a paragraph.
- `.ln(h: float = None)` - Start a new line moving either by the current font height or by the parameter "h". Only permitted when no explicit paragraph is currently active.
- `.render()` - if the region is not used as a context manager with "with", this method must be called to actually process the added text.

Paragraphs

The primary purpose of paragraphs is to enable variations in horizontal text alignment, while the horizontal extents of the text are managed by the text region. To set the alignment, you can use the `align` argument when creating the paragraph. Valid values are defined in the [Align enum](#).

For more typographical control, you can use the following arguments. Most of those override the settings of the current region when set, and default to the value set there.

- `text_align` (Align, optional) - The horizontal alignment of the paragraph.
- `line_height` (float, optional) - factor by which the line spacing will be different from the font height. (default: by region)
- `top_margin` (float, optional) - how much spacing is added above the paragraph. No spacing will be added at the top of the paragraph if the current y position is at (or above) the top margin of the page. (Default: 0.0 mm)
- `bottom_margin` (float, optional) - Those two values determine how much spacing is added below the paragraph. No spacing will be added at the bottom if it would result in overstepping the bottom margin of the page. (Default: 0.0 mm)
- `indent` (float, optional): determines the indentation of the paragraph. (Default: 0.0 mm)
- `bullet_r_margin` (float, optional) - determines the relative displacement of the bullet along the x-axis. The distance is between the rightmost point of the bullet to the leftmost point of the paragraph's text. (Default: 2.0 mm)
- `bullet_string` (str, optional): determines the fragments and text lines of the bullet. (Default: "")
- `skip_leading_spaces` (float, optional) - removes all space characters at the beginning of each line.
- `wrapmode` (WrapMode, optional)

Other than text regions, paragraphs should always be used as context managers and never be reused. Violating those rules may result in the entered text turning up on the page out of sequence.

POSSIBLE FUTURE EXTENSIONS

Those features are currently not supported, but Pull Requests are welcome to implement them:

- per-paragraph indentation
- first-line indentation

Images

New in  2.7.7

Most arguments for inserting images into text regions are the same as for the `FPDF.image()` method, and have the same or equivalent meaning. Since the image will be placed automatically, the "x" and "y" parameters are not available. The positioning can be controlled with "align", where the default is "LEFT", with the alternatives "RIGHT" and "CENTER". If neither width nor height are specified, the image will be inserted with the size resulting from the PDF default resolution of 72 dpi. If the "fill_width" parameter is set to True, it increases the size to fill the full column width if necessary. If the image is wider than the column width, it will always be reduced in size proportionally. The "top_margin" and "bottom_margin" parameters have the same effect as with text paragraphs.

3.10.2 Text Columns

New in  2.7.6

Notice: As of fpdf2 release 2.7.6, this is an experimental feature. Both the API and the functionality may change before it is finalized, without prior notice.

Text Columns

The `FPDF.text_columns()` method allows to create columnar layouts, with one or several columns. Columns will always be of equal width.

PARAMETERS

Beyond the parameters common to all text regions, the following are available for text columns:

- `l_margin` (float, optional) - override the current left page margin.
- `r_margin` (float, optional) - override the current right page margin.
- `ncols` (float, optional) - the number of columns to generate (Default: 2).
- `gutter` (float, optional) - the horizontal space required between each two columns (Default 10).
- `balance` (bool, optional) - Create height balanced columns, starting at the current height and ending at approximately the same level.

METHODS

Text columns support all the standard text region methods like `.paragraph()`, `.write()`, `.ln()`, and `.render()`. In addition to that:

- `.new_column()` - End the current column and continue at the top of the next one.

A FORM_FEED character (`\u000c`) in the text will have the same effect as an explicit call to `.new_column()`,

Note that when used within balanced columns, switching to a new column manually will result in incorrect balancing.

Single-Column Example

In this example an inserted paragraph is used in order to format its content with justified alignment, while the rest of the text uses the default left alignment.

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font("Times", size=12)

cols = pdf.text_columns()
with cols:
    cols.write(text=LOREM_IPSUM[:400])
    with cols.paragraph(
        text_align="J",
        top_margin=pdf.font_size,
        bottom_margin=pdf.font_size
    ) as par:
        par.write(text=LOREM_IPSUM[:400])
    cols.write(text=LOREM_IPSUM[:400])
```

Lorem ipsum Ut nostrud inire reprehenderit anim nostrud dolore sed ut Excepteur dolore ut sunt inire consectetur tempor eu tempor nostrud dolore sint exercitation aliquip velit ullamco esse dolore mollit ea sed voluptate commodo amet eiusmod incididunt Excepteur Excepteur officia est ea dolore sed id in cillum incididunt quis ex id aliqua ullamco reprehenderit cupidatat in quis pariatur ex et veni

Lorem ipsum Ut nostrud inire reprehenderit anim nostrud dolore sed ut Excepteur dolore ut sunt inire consectetur tempor eu tempor nostrud dolore sint exercitation aliquip velit ullamco esse dolore mollit ea sed voluptate commodo amet eiusmod incididunt Excepteur Excepteur officia est ea dolore sed id in cillum incididunt quis ex id aliqua ullamco reprehenderit cupidatat in quis pariatur ex et veni

Lorem ipsum Ut nostrud inire reprehenderit anim nostrud dolore sed ut Excepteur dolore ut sunt inire consectetur tempor eu tempor nostrud dolore sint exercitation aliquip velit ullamco esse dolore mollit ea sed voluptate commodo amet eiusmod incididunt Excepteur Excepteur officia est ea dolore sed id in cillum incididunt quis ex id aliqua ullamco reprehenderit cupidatat in quis pariatur ex et veni

Multi-Column Example

Here we have a layout with three columns. Note that font type and text size can be varied within a text region, while still maintaining the justified (in this case) horizontal alignment.

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font("Helvetica", size=16)

with pdf.text_columns(text_align="j", ncols=3, gutter=5) as cols:
    cols.write(text=LOREM_IPSUM[:600])
    pdf.set_font("Times", "", 18)
    cols.write(text=LOREM_IPSUM[:500])
    pdf.set_font("Courier", "", 20)
    cols.write(text=LOREM_IPSUM[:500])
```



Balanced Columns

Normally the columns will be filled left to right, and if the text ends before the page is full, the rightmost column will be shorter than the others. If you prefer that all columns on a page end on the same height, you can use the `balance=True` argument. In that case a simple algorithm will be applied that attempts to approximately balance their bottoms.

```
from fpdf import FPDF
```

```
pdf = FPDF()
pdf.add_page()
pdf.set_font("Times", size=12)

cols = pdf.text_columns(text_align="J", ncols=3, gutter=5, balance=True)
# fill columns with balanced text
with cols:
    pdf.set_font("Times", "", 14)
    cols.write(text=LOREM_IPSUM[:300])
# add an image below
img_info = pdf.image("../fpdf2/docs/regular_polygon.png",
                    x=pdf.l_margin, w=pdf.epw)
# continue multi-column text
with cols:
    cols.write(text=LOREM_IPSUM[300:600])
```



Note that column balancing only works reliably when the font size (specifically the line height) doesn't change, and if there are no images included. If parts of the text use a larger or smaller font than the rest, then the balancing will usually be out of whack. Contributions for a more refined balancing algorithm are welcome.

POSSIBLE FUTURE EXTENSIONS

Those features are currently not supported, but Pull Requests are welcome to implement them:

- Columns with differing widths (no balancing possible in this case).

3.11 Tables

New in  2.7.0

Tables can be built using the `table()` method. Here is a simple example:

```
from fpdf import FPDF

TABLE_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-Mathurin-sur-Loire"),
)

pdf = FPDF()
pdf.add_page()
pdf.set_font("Times", size=16)
with pdf.table() as table:
    for data_row in TABLE_DATA:
        row = table.row()
        for datum in data_row:
            row.cell(datum)
pdf.output('table.pdf')
```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

3.11.1 Features

- support cells with content wrapping over several lines
- control over column & row sizes (automatically computed by default)
- allow to style table headings (top row), or disable them
- control over borders: color, width & where they are drawn
- handle splitting a table over page breaks, with headings repeated
- control over cell background color
- control table width & position
- control over text alignment in cells, globally or per row
- allow to embed images in cells
- merge cells across columns and rows

3.11.2 Setting table & column widths


```
...
with pdf.table(width=150, col_widths=(30, 30, 10, 30)) as table:
  ...
```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

`align` can be passed to `table()` to set the table horizontal position relative to the page, when it's not using the full page width. It's centered by default.

3.11.3 Setting text alignment

This can be set globally, or on a per-column basis:

```
...
with pdf.table(text_align="CENTER") as table:
  ...
pdf.ln()
with pdf.table(text_align=("CENTER", "CENTER", "RIGHT", "LEFT")) as table:
  ...
```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

3.11.4 Setting cell padding

New in  2.7.6

Cell padding (the space between the cells content and the edge of the cell) can be set globally or on a per-cell basis.

Following the CCS standard the padding can be specified using 1,2 3 or 4 values.

- When one value is specified, it applies the same padding to all four sides.
- When two values are specified, the first padding applies to the top and bottom, the second to the left and right.
- When three values are specified, the first padding applies to the top, the second to the right and left, the third to the bottom.
- When four values are specified, the paddings apply to the top, right, bottom, and left in that order (clockwise)

```
...
style = FontFace(color=black, fill_color=red)
with pdf.table(line_height=pdf.font_size, padding=2) as table:
    for irow in range(5):
        row = table.row()
        for icol in range(5):
            datum = "Circus"
            if irow == 3 and icol % 2 == 0:
                row.cell("custom padding", style=style, padding=(2 * icol, 8, 8, 8))
            else:
                row.cell(datum)
```

(also an example of coloring individual cells)

Circus	Circus	Circus	Circus	Circus
Circus	Circus	Circus	Circus	Circus
Circus	Circus	Circus	Circus	Circus
custom padding	Circus	custom padding	Circus	custom padding
Circus	Circus	Circus	Circus	Circus

Note: the `c_margin` parameter (default 1.0) also controls the horizontal margins in a cell. If a non-zero padding for left and right is supplied then `c_margin` is ignored.

3.11.5 Setting vertical alignment of text in cells

New in  2.7.6

Can be set globally or per cell. Works the same way as padding, but with the `v_align` parameter.

```
with pdf.table(v_align=VAlign.M) as table:
...
    row.cell(f"custom v-align", v_align=VAlign.T) # <-- align to top
```

3.11.6 Setting row height

```
...
with pdf.table(line_height=2.5 * pdf.font_size) as table:
...
```

3.11.7 Disable table headings

By default, `fpdf2` considers that the first row of tables contains its headings. This can however be disabled:

```
...
with pdf.table(first_row_as_headings=False) as table:
...
```

New in  2.7.9

The **repetition** of table headings on every page can also be disabled:

```
...
with pdf.table(repeat_headings=0) as table:
...
```

"ON_TOP_OF EVERY PAGE" is an equivalent valid value for `repeat_headings`, cf. [documentation on TableHeadingsDisplay](#).

3.11.8 Style table headings

```
from fpdf.fonts import FontFace
...
blue = (0, 0, 255)
grey = (128, 128, 128)
headings_style = FontFace(emphasis="ITALICS", color=blue, fill_color=grey)
with pdf.table(headings_style=headings_style) as table:
...
```

Result:

<i>First name</i>	<i>Last name</i>	<i>Age</i>	<i>City</i>
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

It's possible to override the style of individual cells in the heading. The overriding style will take precedence for any specified values, while retaining the default style for unspecified values:

```
...
headings_style = FontFace(emphasis="ITALICS", color=blue, fill_color=grey)
override_style = FontFace(emphasis="BOLD")
with pdf.table(headings_style=headings_style) as table:
    headings = table.row()
    headings.cell("First name", style=override_style)
    headings.cell("Last name", style=override_style)
    headings.cell("Age")
    headings.cell("City")
...
```

Result:

First name	Last name	<i>Age</i>	<i>City</i>
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

3.11.9 Set cells background

```
...
greyscale = 200
with pdf.table(cell_fill_color=greyscale, cell_fill_mode="ROWS") as table:
    ...
```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers


```
...
lightblue = (173, 216, 230)
with pdf.table(cell_fill_color=lightblue, cell_fill_mode="COLUMNS") as table:
    ...
```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

The cell color is set following those settings, ordered by priority:

1. The cell `style`, provided to `Row.cell()`
2. The row `style`, provided to `Table.row()`
3. The table setting `headings_style.fill_color`, if the cell is part of some headings row
4. The table setting `cell_fill_color`, if `cell_fill_mode` indicates to fill a cell
5. The document `.fill_color` set before rendering the table

New in  2.7.9

Finally, it is possible to define your own cell-filling logic:

```
class EvenOddCellFillMode():
    @staticmethod
    def should_fill_cell(i, j):
        return i % 2 and j % 2

...
with pdf.table(cell_fill_color=lightblue, cell_fill_mode=EvenOddCellFillMode()) as table:
    ...
```

3.11.10 Set borders layout

```
...
with pdf.table(borders_layout="INTERNAL") as table:
...
```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

```
...
with pdf.table(borders_layout="MINIMAL") as table:
...
```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

```
...
pdf.set_draw_color(50) # very dark grey
pdf.set_line_width(.5)
with pdf.table(borders_layout="SINGLE_TOP_LINE") as table:
...
```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

All the possible layout values are described there: [TableBordersLayout](#).

3.11.11 Insert images

```
TABLE_DATA = (
    ("First name", "Last name", "Image", "City"),
    ("Jules", "Smith", "shirt.png", "San Juan"),
    ("Mary", "Ramos", "joker.png", "Orlando"),
    ("Carlson", "Banks", "socialist.png", "Los Angeles"),
    ("Lucas", "Cimon", "circle.bmp", "Angers"),
)
pdf = FPDF()
pdf.add_page()
pdf.set_font("Times", size=16)
with pdf.table() as table:
    for i, data_row in enumerate(TABLE_DATA):
        row = table.row()
        for j, datum in enumerate(data_row):
            if j == 2 and i > 0:
                row.cell(img=datum)
            else:
                row.cell(datum)
pdf.output('table_with_images.pdf')
```

Result:

First name	Last name	Image	City
Jules	Smith		San Juan
Mary	Ramos		Orlando
Carlson	Banks		Los Angeles
Lucas	Cimon		Angers

By default, images height & width are constrained by the row height (based on text content) and the column width. To render bigger images, you can set the `line_height` to increase the row height, or pass `img_fill_width=True` to `.cell()`:

```
row.cell(img=datum, img_fill_width=True)
```

Result:

First name	Last name	Image	City
Jules	Smith		San Juan
Mary	Ramos		Orlando
Carlson	Banks		Los Angeles
Lucas	Cimon		Angers

3.11.12 Adding links to cells

```
row.cell(..., link="https://py-pdf.github.io/fpdf2/")
row.cell(..., link=pdf.add_link(page=1))
```

3.11.13 Syntactic sugar

To simplify `table()` usage, shorter, alternative usage forms are allowed.

This sample code:

```
with pdf.table() as table:
    for data_row in TABLE_DATA:
        row = table.row()
```



```
for datum in data_row:
    row.cell(datum)
```

Can be shortened to the following code, by passing lists of strings as the `cells` optional argument of `.row()`:

```
with pdf.table() as table:
    for data_row in TABLE_DATA:
        table.row(data_row)
```

And even shortened further to a single line, by passing lists of lists of strings as the `rows` optional argument of `.table()`:

```
with pdf.table(TABLE_DATA):
    pass
```

3.11.14 Gutter

Spacing can be introduced between rows and/or columns:

```
with pdf.table(TABLE_DATA, gutter_height=3, gutter_width=3):
    pass
```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

3.11.15 Column span and row span

Cells spanning multiple columns or rows can be defined by passing a `colspan` or `rowspan` argument to `.cell()`. Only the cells with data in them need to be defined. This means that the number of cells on each row can be different.

```
...
with pdf.table(col_widths=(1, 2, 1, 1)) as table:
    row = table.row()
    row.cell("0")
    row.cell("1")
    row.cell("2")
    row.cell("3")

    row = table.row()
    row.cell("A1")
    row.cell("A2", colspan=2)
    row.cell("A4")

    row = table.row()
    row.cell("B1", colspan=2)
    row.cell("B3")
    row.cell("B4")
...
```

result:

0	1	2	3
A1	A2		A4
B1		B3	B4

```
...
with pdf.table(text_align="CENTER") as table:
    row = table.row()
    row.cell("A1", colspan=2, rowspan=3)
    row.cell("C1", colspan=2)

    row = table.row()
    row.cell("C2", colspan=2, rowspan=2)

    row = table.row()
    # all columns of this row are spanned by previous rows

    row = table.row()
    row.cell("A4", colspan=4)

    row = table.row()
    row.cell("A5", colspan=2)
    row.cell("C5")
    row.cell("D5")

    row = table.row()
    row.cell("A6")
    row.cell("B6", colspan=2, rowspan=2)
    row.cell("D6", rowspan=2)

    row = table.row()
    row.cell("A7")
...

```

result:

A	B	C	D
A1		C1	
		C2	
A4			
A5		C5	D5
A6	B6		D6
A7			

Alternatively, the spans can be defined using the placeholder elements `TableSpan.COL` and `TableSpan.ROW`. These elements merge the current cell with the previous column or row respectively.

For example, the previous example table can be defined as follows:

```

...
TABLE_DATA = [
    ["A", "B", "C", "D"],
    ["A1", TableSpan.COL, "C1", TableSpan.COL],
    [TableSpan.ROW, TableSpan.ROW, "C2", TableSpan.COL],
    [TableSpan.ROW, TableSpan.ROW, TableSpan.ROW, TableSpan.ROW],
    ["A4", TableSpan.COL, TableSpan.COL, TableSpan.COL],
    ["A5", TableSpan.COL, "C5", "D5"],
    ["A6", "B6", TableSpan.COL, "D6"],
    ["A7", TableSpan.ROW, TableSpan.ROW, TableSpan.ROW],
]

with pdf.table(TABLE_DATA, text_align="CENTER"):
    pass
...

```

result:

A	B	C	D
A1		C1	
		C2	
A4			
A5		C5	D5
A6	B6		D6
A7			

3.11.16 Table with multiple heading rows

The number of heading rows is defined by passing the `num_heading_rows` argument to `Table()`. The default value is `1`. To guarantee backwards compatibility with the `first_row_as_headings` argument, the following applies: - If `num_heading_rows==1`: The value of `first_row_as_headings` defines whether the first row is treated as heading or standard row. - Otherwise, the value of `num_heading_rows` decides the number of heading rows.

```

with pdf.table(TABLE_DATA, num_heading_rows=2):
    pass

```

Result:

First name	Last name	Age	City
Jules	Smith	34	San Juan
Mary	Ramos	45	Orlando
Carlson	Banks	19	Los Angeles
Lucas	Cimon	31	Angers

3.11.17 Table from pandas DataFrame

cf. [Maths documentation page](#)

3.11.18 Using write_html

Tables can also be defined in HTML using `FPDF.write_html`. With the same `data` as above, and column widths defined as percent of the effective width:

```
from fpdf import FPDF

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

Note that `write_html` has [some limitations](#), notably regarding multi-lines cells.

3.11.19 "Parsability" of the tables generated

The PDF file format is not designed to embed structured tables. Hence, it can be tricky to extract tables data from PDF documents.

In our tests suite, we ensure that several PDF-tables parsing Python libraries can successfully extract tables in documents generated with `fpdf2`. Namely, we test [camelot-py](#) & [tabula-py](#): [test/table/test_table_extraction.py](#).

Based on those tests, if you want to ease table extraction from the documents you produce, we recommend the following guidelines:

- avoid splitting tables on several pages
- avoid the `INTERNAL` / `MINIMAL` / `SINGLE_TOP_LINE` borders layouts

4. Text Content

4.1 Adding Text

There are several ways in `fpdf` to add text to a PDF document, each of which comes with its own special features and its own set of advantages and disadvantages. You will need to pick the right one for your specific task.

4.1.1 Simple Text Methods

method	lines	markdown support	HTML support	accepts new current position	details
<code>.text()</code>	one	no	no	fixed	Inserts a single-line text string with a precise location on the base line of the font.
<code>.cell()</code>	one	yes	no	yes	Inserts a single-line text string within the boundaries of a given box, optionally with background and border.
<code>.multi_cell()</code>	several	yes	no	yes	Inserts a multi-line text string within the boundaries of a given box, optionally with background, border and padding.
<code>.write()</code>	several	no	no	auto	Inserts a multi-line text string within the boundaries of the page margins, starting at the current x/y location (typically the end of the last inserted text).
<code>.write_html()</code>	several	no	yes	auto	An extension to <code>.write()</code> , with additional parsing of basic HTML tags.

4.1.2 Flowable Text Regions

Text regions allow to insert flowing text into a predefined region on the page. It is possible to change the formatting and even the font within paragraphs, which will still be aligned as one text block. The currently implemented type of text regions is `text_columns()`, which defines one or several columns that can be filled sequentially or height-balanced.

4.1.3 Typography and Language Specific Concepts

Supported Features

With supporting Unicode fonts, fpdf2 should handle the following text shaping features correctly. More details can be found in [TextShaping](#). * Automatic ligatures / glyph substitution - Some writing systems (eg. most Indic scripts such as Devaganari, Tamil, Kannada) frequently combine a number of written characters into a single glyph. In latin script, "ff", "fi", "ft", "st" and others are often combined. In programming fonts "<=", "++", "!=" etc. may be combined into more compact representations. * Special diacritics that use separate code points (eg. in Diné Bizaad, Hebrew) will be placed in the correct location relative to their base character. * Kerning, where the spacing between characters varies depending on their combination (eg. moving the succeeding lowercase character closer to an uppercase "T". * Left-to-right and right-to-left text formatting (the latter most prominently in Arabic and Hebrew).

Limitations

There are a few advanced typesetting features that fpdf doesn't currently support. * Contextual forms - In some writing systems (eg. Arabic, Mongolian, etc.), characters may take a different shape, depending on whether they appear at the beginning, in the middle, or at the end of a word, or isolated. Fpdf will always use the same standard shape in those cases. * Vertical writing - Some writing systems are meant to be written vertically. Doing so is not directly supported. In cases where this just means to stack characters on top of each other (eg. Chinese, Japanese, etc.), client software can implement this by placing each character individuall at the correct location. In cases where the characters are connected with each other (eg. Mongolian), this may be more difficult, if possible at all.

Character or Word Based Line Wrapping

By default, `multi_cell()` and `write()` will wrap lines based on words, using space characters and soft hyphens as separators. Non-breaking spaces (`\U00a0`) do not trigger a word wrap, but are otherwise treated exactly as a normal space character. For languages like Chinese and Japanese, that don't usually separate their words, character based wrapping is more appropriate. In such a case, the argument `wrapmode="CHAR"` can be used (the default is "WORD"), and each line will get broken right before the character that doesn't fit anymore.

4.1.4 Text Formatting

For all text insertion methods, the relevant font related properties (eg. font/style and foreground/background color) must be set before invoking them. This includes using:

- `.set_font()`
- `.set_text_color()`
- `.set_draw_color()` - for cell borders
- `.set_fill_color()` - for the background

All three `set*_colors()` methods accept either a single greyscale value, 3 values as RGB components, a single `#abc` or `#abcdef` hexadecimal color string, or an instance of `fpdf.drawing.DeviceCMYK`, `fpdf.drawing.DeviceRGB` or `fpdf.drawing.DeviceGray`. You can even use [named web colors](#) by using `html.color_as_decimal()`.

More text styling options can be found on the page [Text styling](#), including [Markdown syntax](#) and [HTML markup](#).

4.1.5 Change in current position

`.cell()` and `.multi_cell()` let you specify where the current position (`.x / .y`) should go after the call. This is handled by the parameters `new_x` and `new_y`. Their values must one of the following enums values or an equivalent string:

- `XPos`
- `YPos`

4.1.6 .text()

Prints a single-line character string. In contrast to the other text methods, the position is given explicitly, and not taken from `.x / .y`. The origin is on the left of the first character, on the baseline. This method allows placing a string with typographical precision on the page, but it is usually easier to use the `.cell()`, `.multi_cell()` or `.write()` methods.

[Signature and parameters for .text\(\)](#)

4.1.7 .cell()

Prints a cell (rectangular area) with optional borders, background color and character string. The upper-left corner of the cell corresponds to the current position. The text can be aligned or centered. After the call, the current position moves to the selected `new_x / new_y` position. It is possible to put a link on the text. If `markdown=True`, then minimal [markdown](#) styling is enabled, to render parts of the text in bold, italics, and/or underlined.

If automatic page breaking is enabled and the cell goes beyond the limit, a page break is performed before outputting.

[Signature and parameters for.cell\(\)](#)

4.1.8 .multi_cell()

Allows printing text with word or character based line breaks. Those can be automatic (breaking at the most recent space or soft-hyphen character) as soon as the text reaches the right border of the cell, or explicit (via the `\n` character). As many cells as necessary are stacked, one below the other. Text can be aligned, centered or justified. The cell block can be framed and the background painted. Padding between text and the cell edge can be specified in the same way as for tables.

Using `new_x="RIGHT", new_y="TOP", maximum height=pdf.font_size` can be useful to build tables with multiline text in cells.

In normal operation, returns a boolean indicating if page break was triggered. The return value can be altered by specifying the `output` parameter.

[Signature and parameters for.multi_cell\(\)](#)

4.1.9 .write()

Prints multi-line text between the page margins, starting from the current position. When the right margin is reached, a line break occurs at the most recent space or soft-hyphen character (in word wrap mode) or at the current position (in character break mode), and text continues from the left margin. A manual break happens any time the `\n` character is met. Upon method exit, the current position is left near the end of the text, ready for the next call to continue without a gap, potentially with a different font or size set. Returns a boolean indicating if page break was triggered.

The primary purpose of this method is to print continuously wrapping text, where different parts may be rendered in different fonts or font sizes. This contrasts eg. with `.multi_cell()`, where a change in font family or size can only become effective on a new line.

[Signature and parameters for.write\(\)](#)

4.1.10 .write_html()

This method is very similar to `.write()`, but accepts basic HTML formatted text as input. See [html.py](#) for more details and the supported HTML tags.

Note that when using data from actual web pages, the result may not look exactly as expected, because `.write_html()` prints all whitespace unchanged as it finds them, while webbrowsers rather collapse each run of consecutive whitespace into a single space character.

[Signature and parameters for .write_html\(\)](#)

4.2 Line breaks

When using `multi_cell()` or `write()`, 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 line 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.

If the parameter `print_sh=False` in `multi_cell()` or `write()` is set to `True`, then they will print the soft-hyphen character to the document (as a normal hyphen with most fonts) instead of using it as a line break opportunity.

4.3 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.

4.3.1 Manually trigger a page break

Simply call `.add_page()`.

4.3.2 Inserting the final number of pages of the document

The special string `{nb}` will be substituted by the total number of pages on document closure. This special value can be changed by calling `alias_nb_pages()`.

4.3.3 will_page_break

`will_page_break(height)` lets you know if adding an element will trigger a page break, based on its `height` and the current ordinate (`y` position).

4.3.4 Unbreakable sections

In order to render content, like `tables`, with the insurance that no page break will be performed in it, one 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 doc:
        for row in data: # data comes from snippets on the Tables documentation page
            for datum in row:
                doc.cell(col_width, line_height, f"{datum} ({i})", border=1)
            doc.ln(line_height)
        print('page_break_triggered:', doc.page_break_triggered)
        pdf.ln(line_height * 2)
pdf.output("unbreakable_tables.pdf")
```

An alternative approach is `offset_rendering()` that allows to test the results of some operations on the global layout before performing them "for real":

```
with pdf.offset_rendering() as dummy:
    # Dummy rendering:
    dummy.multi_cell(...)
if dummy.page_break_triggered:
    # We trigger a page break manually beforehand:
    pdf.add_page()
    # We duplicate the section header:
    pdf.cell(text="Appendix C")
# Now performing our rendering for real:
pdf.multi_cell(...)
```

4.4 Text styling

4.4.1 .set_font()

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

- `style="B"` indicates **bold**
- `style="I"` indicates *italics*
- `style="U"` indicates underline
- `style="BI"` indicates ***bold italics***

```
from fpdf import FPDF

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

4.4.2 .set_stretching(stretching=100)

Text can be stretched horizontally with this setting, measured in percent. If the argument is less than 100, then all characters are rendered proportionally narrower and the text string will take less space. If it is larger than 100, then the width of all characters will be expanded accordingly.

The example shows the same text justified to the same width, with stretching values of 100 and 150.

```
pdf = FPDF()
pdf.add_page()
pdf.set_font("Helvetica", "", 8)
pdf.set_fill_color(255, 255, 0)
pdf.multi_cell(w=50, text=LOREM_IPSUM[:100], new_x="LEFT", fill=True)
pdf.ln()
pdf.set_stretching(150)
pdf.multi_cell(w=50, text=LOREM_IPSUM[:100], new_x="LEFT", fill=True)
```

Lorem ipsum Ut nostrud irure
reprehenderit anim nostrud dolore sed
ut Excepteur dolore ut sunt irure

Lorem ipsum Ut nostrud
irure reprehenderit anim
nostrud dolore sed ut
Excepteur dolore ut sunt
irure

4.4.3 .set_char_spacing(spacing=0)

This method changes the distance between individual characters of a text string. Normally, characters are placed at a given distance according to the width information in the font file. If spacing is larger than 0, then their distance will be larger, creating a gap in between. If it is less than 0, then their distance will be smaller, possibly resulting in an overlap. The change in distance is given in typographic points (Pica), which makes it easy to adapt it relative to the current font size.

Character spacing works best for formatting single line text created by any method, or for highlighting individual words included in a block of text with `.write()`.

Limitations: Spacing will only be changed *within* a sequence of characters that `fpdf2` adds to the PDF in one go. This means that there will be no extra distance *eg.* between text parts that are placed successively with `write()`. Also, if you apply different

font styles using the Markdown functionality of `.cell()` and `.multi_cell()` or by using `html_write()`, then any parts given different styles will have the original distance between them. This is so because `fpdf2` has to add each styled fragment to the PDF file separately.

The example shows the same text justified to the same width, with `char_spacing` values of 0 and 10 (font size 8 pt).

```
pdf = FPDF()
pdf.add_page()
pdf.set_font("Helvetica", "", 8)
pdf.set_fill_color(255, 255, 0)
pdf.multi_cell(w=150, text=LOREM_IPSUM[:200], new_x="LEFT", fill=True)
pdf.ln()
pdf.set_char_spacing(10)
pdf.multi_cell(w=150, text=LOREM_IPSUM[:200], new_x="LEFT", fill=True)
```

Lorem ipsum Ut nostrud irure reprehenderit anim nostrud dolore sed ut Excepteur dolore ut sunt irure consectetur tempor eu tempor nostrud dolore sint exercitation aliquip velit ullamco esse dolore mol

L o r e m i p s u m U t n o s t r u d i r u r e
r e p r e h e n d e r i t E x c e p t e u r d o l o r e
d o l o r e s e d u t E x c e p t e u r d o l o r e
u t s u n t i r u r e c o n s e c t e t u r
t e m p o r e u t e m p o r n o s t r u d
d o l o r e s i n t e x e r c i t a t i o n
a l i q u i p v e l i t u l l a m c o e s s e
d o l o r e m o l

For a more complete support of **Markdown** syntax, check out this guide to combine `fpdf2` with the `mistletoe` library: [Combine with mistletoe to use Markdown](#).

4.4.4 Subscript, Superscript, and Fractional Numbers

The class attribute `.char_vpos` controls special vertical positioning modes for text:

- "LINE" - normal line text (default)
- "SUP" - superscript (exponent)
- "SUB" - subscript (index)
- "NOM" - nominator of a fraction with "/"
- "DENOM" - denominator of a fraction with "/"

For each positioning mode there are two parameters that can be configured. The defaults have been set to result in a decent layout with most fonts, and are given in parens.

The size multiplier for the font size:

- `.sup_scale` (0.7)
- `.sub_scale` (0.7)
- `.nom_scale` (0.75)
- `.denom_scale` (0.75)

The lift is given as fraction of the unscaled font size and indicates how much the glyph gets lifted above the base line (negative for below):

- `.sup_lift` (0.4)
- `.sub_lift` (-0.15)
- `.nom_lift` (0.2)
- `.denom_lift` (0.0)

Limitations: The individual glyphs will be scaled down as configured. This is not typographically correct, as it will also reduce the stroke width, making them look lighter than the normal text. Unicode fonts may include characters in the [subscripts and superscripts range](#). In a high quality font, those glyphs will be smaller than the normal ones, but have a proportionally stronger stroke width in order to maintain the same visual density. If available in good quality, using Characters from this range is

preferred and will look better. Unfortunately, many fonts either don't (fully) cover this range, or the glyphs are of unsatisfactory quality. In those cases, this feature of `fpdf2` offers a reliable workaround with suboptimal but consistent output quality.

Practical use is essentially limited to `.write()` and `html_write()`. The feature does technically work with `.cell()` and `.multi_cell`, but is of limited usefulness there, since you can't change font properties in the middle of a line (there is no markdown support). It currently gets completely ignored by `.text()`.

The example shows the most common use cases:

```
pdf = fpdf.FPDF()
pdf.add_page()
pdf.set_font("Helvetica", "", 20)
pdf.write(text="2")
pdf.char_vpos = "SUP"
pdf.write(text="56")
pdf.char_vpos = "LINE"
pdf.write(text=" more line text")
pdf.char_vpos = "SUB"
pdf.write(text="(idx)")
pdf.char_vpos = "LINE"
pdf.write(text=" end")
pdf.ln()
pdf.write(text="1234 + ")
pdf.char_vpos = "NOM"
pdf.write(text="5")
pdf.char_vpos = "LINE"
pdf.write(text="/")
pdf.char_vpos = "DENOM"
pdf.write(text="16")
pdf.char_vpos = "LINE"
pdf.write(text=" + 987 = x")
```

2⁵⁶ more line text_(idx) end
1234 + ⁵/₁₆ + 987 = x

4.4.5 .text_mode

The PDF spec defines several text modes:

TABLE 5.3 Text rendering modes

MODE	EXAMPLE	DESCRIPTION
0		Fill text.
1		Stroke text.
2		Fill, then stroke text.
3		Neither fill nor stroke text (invisible).
4		Fill text and add to path for clipping (see above).
5		Stroke text and add to path for clipping.
6		Fill, then stroke text and add to path for clipping.
7		Add text to path for clipping.

The text mode can be controlled with the `.text_mode` attribute. With `STROKE` modes, the line width is induced by `.line_width`, and its color can be configured with `set_draw_color()`. With `FILL` modes, the filling color can be controlled by `set_fill_color()` or `set_text_color()`.

With any of the 4 `CLIP` modes, the letters will be filled by vector drawings made afterwards, as can be seen in this example:

```
from fpdf import FPDF

pdf = FPDF(orientation="landscape")
pdf.add_page()
pdf.set_font("Helvetica", size=100)

with pdf.local_context(text_mode="STROKE", line_width=2):
    pdf.cell(text="Hello world")
# Outside the local context, text_mode & line_width are reverted
```

```
# back to their original default values
pdf.ln()

with pdf.local_context(text_mode="CLIP"):
    pdf.cell(text="CLIP text mode")
    for r in range(0, 250, 2): # drawing concentric circles
        pdf.circle(x=130-r/2, y=70-r/2, r=r)

pdf.output("text-modes.pdf")
```



More examples from [test_text_mode.py](#):

- [text_modes.pdf](#)
- [clip_text_modes.pdf](#)

4.4.6 markdown=True

An optional `markdown=True` parameter can be passed to the `cell()` & `multi_cell()` methods in order to enable basic Markdown-like styling: `bold`, `italics`, `--underlined--`.

Bold & italics require using dedicated fonts for each style.

For the standard fonts (Courier, Helvetica & Times), those dedicated fonts are configured by default:

```
from fpdf import FPDF

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

Using other fonts means that their variants (bold, italics) must be registered using `add_font` with `style="B"` and `style="I"`. Several unit tests in `test/text/` demonstrate that:

- [test_cell_markdown_with_ttf_fonts](#)
- [test_multi_cell_markdown_with_ttf_fonts](#)

4.4.7 .write_html()

`.write_html()` allows to set emphasis on text through the ``, `<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>""")
```

4.5 Fonts and Unicode

Besides the limited set of latin fonts built into the PDF format, `fpdf2` offers full support for using and embedding Unicode (TrueType "ttf" and OpenType "otf") fonts. To keep the output file size small, it only embeds the subset of each font that is actually used in the document. This part of the code has been completely rewritten since the fork from PyFPDF. It uses the [fonttools](#) library for parsing the font data, and [harfbuzz](#) (via [uharfbuzz](#)) for [text shaping](#).

To make use of that functionality, you have to install at least one Unicode font, either in the system font folder or in some other location accessible to your program. For professional work, many designers prefer commercial fonts, suitable to their specific needs. There are also many sources of free TTF fonts that can be downloaded online and used free of cost (some of them may have restrictions on commercial redistribution, such as server installations or including them in a software project).

- [Font Library](#) - A collection of fonts for many languages with an open source type license.
- [Google Fonts](#) - A collection of free to use fonts for many languages.
- [Microsoft Font Library](#) - A large collection of fonts that are free to use.
- [GitHub: Fonts](#) - Links to public repositories of open source font projects as well as font related software projects.
- [GNU FreeFont](#) family: FreeSans, FreeSerif, FreeMono

To use a Unicode font in your program, use the `add_font()`, and then the `set_font()` method calls.

Built-in Fonts vs. Unicode Fonts

The PDF file format knows a small number of "standard" fonts, namely **Courier**, **Helvetica**, **Times**, **Symbol**, and **ZapfDingbats**. The first three are available in regular, bold, italic, and bold-italic versions. This gives us a set of fonts known as "14 Standard PDF fonts". Any PDF processor (eg. a viewer) must provide those fonts for display. To use them, you don't need to call `.add_font()`, but only `.set_font()`.

Courier

Courier Bold

Courier Italics

Courier Bold Italics

Helvetica

Helvetica Bold

Helvetica Italics

Helvetica Bold Italics

Times

Times Bold

Times Italics

Times Bold Italics

Symbol : Σψμβολ

Zapfdingbats : 

(script used to generate this: [tutorial/core_fonts.py](#))

While that may seem convenient, there's a big drawback. Those fonts only support latin characters, or a set of special characters for the last two. If you try to render any Unicode character outside of those ranges, then you'll get an error like:

"Character "θ" at index 13 in text is outside the range of characters supported by the font used: "courier". Please consider using a Unicode font.". So if you want to create documents with any characters other than those common in English and a small number of european languages, then you need to add a Unicode font containing the respective glyph as described in this document.

Note that even if you have a font eg. named "Courier" installed as a system font on your computer, by default this will not be used. You'll have to explicitly call eg. `.add_font("Courier2", fname=r"C:\Windows\Fonts\cour.ttf")` to make it available. If the name is really the same (ignoring case), then you'll have to use a suitable variation, since trying to overwrite one of the "standard" names with `.add_font()` will result in an error.

Adding and Using Fonts

Before using a Unicode font, you need to load it from a font file. Usually you'll have call `add_font()` for each style of the same font family you want to use. The styles that fpdf2 understands are:

- Regular: ""
- Bold: "b"
- Italic/Oblique: "i"
- Bold-Italic: "bi"

Note that we use the same family name for each of them, but load them from different files. Only when a font has variants (eg. "narrow"), or there are more styles than the four standard ones (eg. "black" or "extra light"), you'll have to add those with a different family name. If the font files are not located in the current directory, you'll have to provide a file name with a relative or absolute path. If the font is not found elsewhere, then fpdf2 will look for it in a subdirectory named "font".

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
# Different styles of the same font family.
pdf.add_font("dejavu-sans", style="", fname="DejaVuSans.ttf")
pdf.add_font("dejavu-sans", style="b", fname="DejaVuSans-Bold.ttf")
pdf.add_font("dejavu-sans", style="i", fname="DejaVuSans-Oblique.ttf")
pdf.add_font("dejavu-sans", style="bi", fname="DejaVuSans-BoldOblique.ttf")
# Different type of the same font design.
pdf.add_font("dejavu-sans-narrow", style="", fname="DejaVuSansCondensed.ttf")
pdf.add_font("dejavu-sans-narrow", style="i", fname="DejaVuSansCondensed-Oblique.ttf")
```

To actually use the loaded font, or to use one of the standard built-in fonts, you'll have to set the current font before calling any text generating method. `set_font()` uses the same combinations of family name and style as arguments, plus the font size in typographic points. In addition to the previously mentioned styles, the letter "u" may be included for creating underlined text. If the family or size are omitted, the already set values will be retained. If the style is omitted, it defaults to regular.

```
# Set and use first family in regular style.
pdf.set_font(family="dejavu-sans", style="", size=12)
pdf.cell(text="Hello")
# Set and use the same family in bold style.
pdf.set_font(style="b", size=18) # still uses the same dejavu-sans font family.
pdf.cell(text="Fat World")
# Set and use a variant in italic and underlined.
pdf.set_font(family="dejavu-sans-narrow", style="iu", size=12)
pdf.cell(text="lean on me")
```

Hello **Fat World** *lean on me*

Note on non-latin languages

Many non-latin writing systems have complex ways to combine characters, ligatures, and possibly multiple diacritic symbols together, change the shape of characters depending on its location in a word, or use a different writing direction. A small number of examples are:

- Hebrew - right-to-left, placement of diacritics
- Arabic - right-to-left, contextual shapes
- Thai - stacked diacritics
- Devanagari (and other indic scripts) - multi-character ligatures, reordering

To make sure those scripts to be rendered correctly, [text shaping](#) must be enabled with `set_text_shaping(True)`.

Right-to-Left scripts

When [text shaping](#) is enabled, fpdf2 will apply the [Unicode Bidirectional Algorithm](#) to render correctly any text, including bidirectional (mix of right-to-left and left-to-right scripts).

4.5.1 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()
pdf.set_text_shaping(True)

# 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(fname='DejaVuSansCondensed.ttf')
pdf.set_font('DejaVuSansCondensed', 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 העולם
"""

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(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(fname='fireflysung.ttf')
pdf.set_font('fireflysung', size=14)
pdf.write(8, u'Chinese: 你好世界\n')
pdf.write(8, u'Japanese: こんにちは世界\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(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(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](#)

4.5.2 Free Font Pack

For your convenience, the author of the original PyFPDF has collected 96 TTF files in an optional "[Free Unicode TrueType Font Pack for FPDF](#)", with useful fonts commonly distributed with GNU/Linux operating systems. Note that this collection is from 2015, so it will not contain any newer fonts or possible updates.

4.5.3 Fallback fonts

New in  2.7.0

The method `set_fallback_fonts()` allows you to specify a list of fonts to be used if any character is not available on the font currently set. When a character doesn't exist on the current font, `fpdf2` will look if it's available on the fallback fonts, on the same order the list was provided.

Common scenarios are use of special characters like emojis within your text, greek characters in formulas or citations mixing different languages.

Example:

```
import fpdf

pdf = fpdf.FPDF()
pdf.add_page()
pdf.add_font(fname="Roboto.ttf")
# twitter emoji font: https://github.com/13rac1/twemoji-color-font/releases
pdf.add_font(fname="TwitterEmoji.ttf")
pdf.set_font("Roboto", size=15)
pdf.set_fallback_fonts(["TwitterEmoji"])
pdf.write(text="text with an emoji 🍌")
pdf.output("text_with_emoji.pdf")
```

When a glyph cannot be rendered using the current font, `fpdf2` will look for a fallback font matching the current character emphasis (bold/italics). By default, if it does not find such matching font, the character will not be rendered using any fallback font. This behaviour can be relaxed by passing `exact_match=False` to `set_fallback_fonts()`.

Moreover, for more control over font fallback election logic, the `get_fallback_font()` can be overridden. An example of this can be found in [test/fonts/test_font_fallback.py](#).

4.6 Text Shaping

New in  2.7.5

4.6.1 What is text shaping?

Text shaping is a fundamental process in typography and computer typesetting that influences the aesthetics and readability of text in various languages and scripts. It involves the transformation of Unicode text into glyphs, which are then positioned for display or print.

For texts in latin script, text shaping can improve the aesthetics by replacing characters that would colide or overlap by a single glyph specially crafted to look harmonious.

Without shaping:
different final floating stuff

With shaping:
different final floating stuff

This process is especially important for scripts that require complex layout, such as Arabic or Indic scripts, where characters change shape depending on their context.

There are three primary aspects of text shaping that contribute to the overall appearance of the text: kerning, ligatures, and glyph substitution.

Kerning

Kerning refers to the adjustment of space between individual letter pairs in a font. This process is essential to avoid awkward gaps or overlaps that may occur due to the default spacing of the font. By manually or programmatically modifying the kerning, we can ensure an even and visually pleasing distribution of letters, which significantly improves the readability and aesthetic quality of the text.

Without shaping:
T,T,T,T,T,T,

With shaping:
T,T,T,T,T,T,

Ligatures

Ligatures are special characters that are created by combining two or more glyphs. This is frequently used to avoid collision between characters or to adhere to the typographic traditions. For instance, in English typography, the most common ligatures are "fi" and "fl", which are often fused into single characters to provide a more seamless reading experience.

Glyph Substitution

Glyph substitution is a mechanism that replaces one glyph or a set of glyphs with one or more alternative glyphs. This is a crucial aspect of text shaping, especially for complex scripts where the representation of a character can significantly vary based on its surrounding characters. For example, in Arabic script, a letter can have different forms depending on whether it's at the beginning, middle, or end of a word.

Another common use of glyph substitution is to replace a sequence of characters by a symbol that better represent the meaning of those characters on a specialized context (mathematical, programming, etc.).

Without shaping:
/* www 0xFF c++ --> */

With shaping:
/* www 0×FF c++ → */

4.6.2 Usage

Text shaping is disabled by default to keep backwards compatibility, reduce resource requirements and not make uharfbuzz a hard dependency.

If you want to use text shaping, the first step is installing the uharfbuzz package via pip.

```
pip install uharfbuzz
```

⚠ Text shaping is *not* available for type 1 fonts.

Basic usage

The method `set_text_shaping()` is used to control text shaping on a document. The only mandatory argument, `use_shaping_engine` can be set to `True` to enable the shaping mechanism or `False` to disable it.

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.add_font(family="ViaodaLibre", fname=HERE / "ViaodaLibre-Regular.ttf")
pdf.set_font("ViaodaLibre", size=40)
pdf.set_text_shaping(True)
pdf.cell(text="final soft stuff")
pdf.output("Example.pdf")
```

Features

On most languages, Harfbuzz enables all features by default. If you want to enable or disable a specific feature you can pass a dictionary containing the 4 digit OpenType feature code as key and a boolean value to indicate if it should be enabled or disabled.

Example:

```
pdf.set_text_shaping(use_shaping_engine=True, features={"kern": False, "liga": False})
```

The full list of OpenType feature codes can be found [here](#)

Additional options

To perform the text shaping, harfbuzz needs to know some information like the language and the direction (right-to-left, left-to-right, etc) in order to apply the correct rules. Those information can be guessed based on the text being shaped, but you can also set the information to make sure the correct rules will be applied.

Examples:

```
pdf.set_text_shaping(use_shaping_engine=True, direction="rtl", script="arab", language="ara")
```

```
pdf.set_text_shaping(use_shaping_engine=True, direction="ltr", script="latn", language="eng")
```

Direction can be `ltr` (left to right) or `rtl` (right to left). The `ttb` (top to bottom) and `btt` (bottom to top) directions are not supported by fpdf2 for now.

[Valid OpenType script tags](#)

[Valid OpenType language codes](#)

4.7 Bidirectional Text

New in  2.7.8

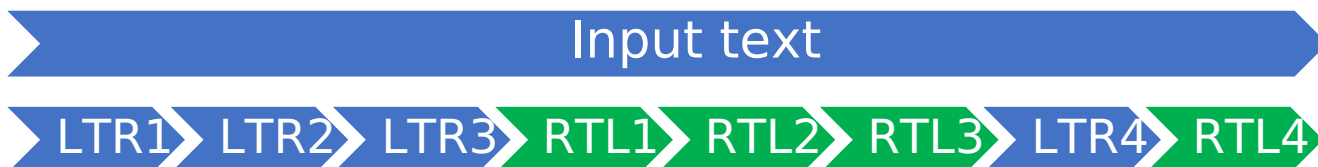
Bidirectional text refers to text containing both left-to-right (LTR) and right-to-left (RTL) language scripts. Languages such as Arabic, Hebrew, and Persian are written from right to left, whereas languages like English, Spanish, and French are written from left to right. The Unicode Bidirectional Algorithm is a set of rules defined by the Unicode Consortium to properly display mixed-directional text. This algorithm ensures that characters are shown in their correct order, preserving the logical sequence of the text.

4.7.1 Unicode Bidirectional Algorithm

The Unicode Bidirectional Algorithm, often abbreviated as the *Bidi* Algorithm, is essential for displaying text containing both RTL and LTR scripts. It determines the directionality of characters and arranges them in a visually correct order. This algorithm takes into account the inherent directionality of characters (such as those in Arabic or Hebrew being inherently RTL) and the surrounding context to decide how text should be displayed.

4.7.2 Paragraph direction

First step- reading the input text and splitting it into directional segments according to the Unicode bidirectional algorithm and fragment characteristics (font, style, etc.)



Second step- the fragments are regrouped into directional runs and shaped with HarfBuzz and a paragraph is built, line by line according to the paragraph direction

Example with paragraph direction left to right



Example with paragraph direction right to left:



4.7.3 Bidirectional text in fpdf2

fpdf2 will automatically apply the unicode bidirectional algorithm if text shaping is enabled.

If no `direction` parameter is provided - or `direction` is `None` - paragraph direction will be set according to the first directional character present on the text.

If there is a need to explicitly set the direction of a paragraph, regardless of the content, you can force the paragraph direction to either RTL or LTR.

```
fpdf.set_text_shaping(use_shaping_engine=True, direction="rtl")
```


4.8 Emojis, Symbols & Dingbats

- [Emojis, Symbols & Dingbats](#)
- [Emojis](#)
- [Symbols](#)
- [Dingbats](#)
- [Fallback fonts](#)

4.8.1 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(fname="DejaVuSans.ttf")
pdf.set_font("DejaVuSans", size=64)
pdf.add_page()
pdf.multi_cell(0, text="".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](#)

Another font supporting emojis is: [twemoji](#)

4.8.2 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, text="\u0022 \u0068 \u0024 \u0065 \u00ce \u00c2, \u0068/\u0065 \u0040 \u00a5",
        new_x="LMARGIN", new_y="NEXT")
pdf.cell(h=16, text="\u0044 \u0046 \u0053 \u0057 \u0059 \u0061 \u0062 \u0063",
        new_x="LMARGIN", new_y="NEXT")
pdf.cell(h=16, text="\u00a0 \u00a7 \u00a8 \u00a9 \u00aa \u00ab \u00ac \u00ad \u00ae \u00af \u00db \u00dc \u00de",
        new_x="LMARGIN", new_y="NEXT")
pdf.output("symbol.pdf")
```

This results in:

$\forall \eta \exists \varepsilon \in \mathbb{R}, \eta/\varepsilon \cong \infty$
 $\Delta \Phi \Sigma \Omega \Psi \alpha \beta \chi$
 € ♣ ♦ ♥ ♠ ↔ ← ↑ → ↓ ⇔ ⇐ ⇒

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](#).

4.8.3 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(text="+ 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](#).

4.8.4 Fallback fonts

If you need to mix special characters and emojis within normal text, it is possible to specify alternative fonts for FPDF to use as fallback fonts. See an example of use [Here](#)

4.9 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 **not** supported, and neither is CSS, but bug reports & contributions are very welcome to improve this. *cf.* [Supported HTML features](#) below for details on its current limitations.

For a more robust & feature-full HTML-to-PDF converter in Python, you may want to check [Reportlab](#) (or [xhtml2pdf](#) based on it), [WeasyPrint](#) or [borb](#).


4.9.1 write_html usage example

HTML rendering requires the use of `FPDF.write_html()` :

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.write_html("""
<dl>
  <dt>Description title</dt>
  <dd>Description Detail</dd>
</dl>
<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/py-pdf/fpdf2">py-pdf/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 type="circle">
    <li>unordered</li>
    <li>list</li>
    <li>items</li>
  </ul>
  <ol start="3" type="i">
    <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")
```

Styling HTML tags globally

New in  2.7.9

The style of several HTML tags (`<a>`, `<blockquote>`, `<code>`, `<pre>`, `<h1>`, `<h2>`, `<h3>` ...) can be set globally, for the whole HTML document, by passing `tag_styles` to `FPDF.write_html()`:

```
from fpdf import FPDF, FontFace

pdf = FPDF()
pdf.add_page()
pdf.write_html("""
<h1>Big title</h1>
<section>
  <h2>Section title</h2>
  <p>Hello world!</p>
</section>
""", tag_styles={
  "h1": FontFace(color=(148, 139, 139), size_pt=32),
  "h2": FontFace(color=(148, 139, 139), size_pt=24),
})
pdf.output("html_styled.pdf")
```

Similarly, the indentation of several HTML tags (`<blockquote>`, `<dd>`, ``) can be set globally, for the whole HTML document, by passing `tag_indents` to `FPDF.write_html()`:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.write_html("""
<dl>
  <dt>Term</dt>
  <dd>Definition</dd>
</dl>
""", tag_indents={"dd": 5})
pdf.output("html_dd_indented.pdf")
```

4.9.2 Supported HTML features

- `<h1>` to `<h8>`: headings (and `align` attribute)
- `<p>`: paragraphs (and `align`, `line-height` attributes)
- `
` & `<hr>` tags
- ``, `<i>`, `<u>`: bold, italic, underline
- ``: (and `face`, `size`, `color` attributes)
- `<center>` for aligning
- `<a>`: links (and `href` attribute) to a file, URL, or page number.
- `<pre>` & `<code>` tags
- ``: images (and `src`, `width`, `height` attributes)
- ``, ``, ``: ordered, unordered and list items (can be nested)
- `<dl>`, `<dt>`, `<dd>`: description list, title, details (can be nested)
- `<sup>`, `<sub>`: superscript and subscript text
- `<table>`: (with `align`, `border`, `width`, `cellpadding`, `cellspacing` attributes)
- `<thead>`: optional tag, wraps the table header row
- `<tfoot>`: optional tag, wraps the table footer row
- `<tbody>`: optional tag, wraps the table rows with actual content
- `<tr>`: rows (with `align`, `bgcolor` attributes)
- `<th>`: heading cells (with `align`, `bgcolor`, `width` attributes)
- `<td>`: cells (with `align`, `bgcolor`, `width`, `rowspan`, `colspan` attributes)

Page breaks

New in  2.7.10

Page breaks can be triggered explicitly using the [break-before](#) or [break-after](#) CSS properties. For example you can use:

```
<br style="break-after: page">
```

or:

```
<p style="break-before: page">
Top of a new page.
</p>
```

4.9.3 Known limitations

fpdf2 HTML renderer does not support some configurations of nested tags. For example:

- `<table>` cells can contain `<td>nested tags forming a single text block</td>`, but **not** `<td>arbitrarilynested tags</td>` - cf. [issue #845](#)

You can also check the currently open GitHub issues with the tag `html`: [label:html is:open](#)

4.9.4 Using Markdown

Check [Combine with mistletoe to use Markdown](#)

5. Graphics Content

5.1 Images

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

- explicit width and height (expressed in user units). The image is scaled to those dimensions, unless `keep_aspect_ratio=True` is specified.
- 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.

5.1.1 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")
```

By default an image is rendered with a resolution of 72 dpi, but you can control its dimension on the page using the `w=` & `h=` parameters of the `image()` method.

5.1.2 Alpha / transparency

`fpdf2` allows to embed images with alpha pixels.

Technically, it is implemented by extracting an `/SMask` from images with transparency, and inserting it along with the image data in the PDF document. Related code is in the `image_parsing` module.

5.1.3 Assembling images

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

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")
```

Fitting an image inside a rectangle

When you want to scale an image to fill a rectangle, while keeping its aspect ratio, and ensuring it does **not** overflow the rectangle width nor height in the process, you can set `w / h` and also provide `keep_aspect_ratio=True` to the `image()` method.

The following unit tests illustrate that:

- `test_image_fit.py`
- resulting document: `image_fit_in_rect.pdf`

Blending images

You can control the color blending mode of overlapping images. Valid values for `blend_mode` are `Normal`, `Multiply`, `Screen`, `Overlay`, `Darken`, `Lighten`, `ColorDodge`, `ColorBurn`, `HardLight`, `SoftLight`, `Difference`, `Exclusion`, `Hue`, `Saturation`, `Color` and `Luminosity`.

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.image("imgA.png", ...)
with pdf.local_context(blend_mode="ColorBurn"):
    pdf.image("imgB.jpg", ...)
pdf.output("blended-images.pdf")
```

Demo of all color blend modes: [blending_images.pdf](#)

5.1.4 Image clipping



You can select only a portion of the image to render using clipping methods:

- `rect_clip()` :
 - [example code](#)
 - [resulting PDF](#)
- `round_clip()` :
 - [example code](#)
 - [resulting PDF](#)
- `elliptic_clip()` :
 - [example code](#)
 - [resulting PDF](#)

5.1.5 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")
```

5.1.6 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")
```

5.1.7 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")
```

5.1.8 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")
```

5.1.9 Image compression

By default, `fpdf2` will avoid altering or recompressing your images: when possible, the original bytes from the JPG or TIFF file will be used directly. Bitonal images are by default compressed as TIFF Group4.

However, you can easily tell `fpdf2` to embed all images as JPEGs in order to reduce your PDF size, using `set_image_filter()`:

```
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 into JPEGs this way will fill transparent areas of your images with color (usually black).

The allowed `image_filter` values are listed in the [image_parsing](#) module and are currently: `FlateDecode` (lossless zlib/deflate compression), `DCTDecode` (lossy compression with JPEG) and `JPXDecode` (lossy compression with JPEG2000).

5.1.10 ICC Profiles

The ICC profile of the included images are read through the PIL function `Image.info.get("icc_profile")` and are included in the PDF as objects.

5.1.11 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](#)

5.1.12 Disabling transparency

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

```
from fpdf import FPDF

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

This will fill transparent areas of your images with color (usually black).

cf. also documentation on [controlling transparency](#).

5.1.13 Page background

cf. [Per-page format, orientation and background](#)

5.1.14 Sharing the image cache among FPDF instances

Image parsing is often the most CPU & memory intensive step when inserting pictures in a PDF.

If you create several PDF files that use the same illustrations, you can share the images cache among FPDF instances:

```
image_cache = None

for ... # loop
    pdf = FPDF()
    if image_cache is None:
        image_cache = pdf.image_cache
    else:
        pdf.image_cache = image_cache
    ... # build the PDF
    pdf.output(...)
    # Reset the "usages" count, to avoid ALL images to be inserted in subsequent PDFs:
    image_cache.reset_usages()
```

This recipe is valid for `fpdf2` v2.5.7+. For previous versions of `fpdf2`, a *deepcopy* of `.images` must be made, (*cf.* [issue #501](#)).

5.2 Shapes

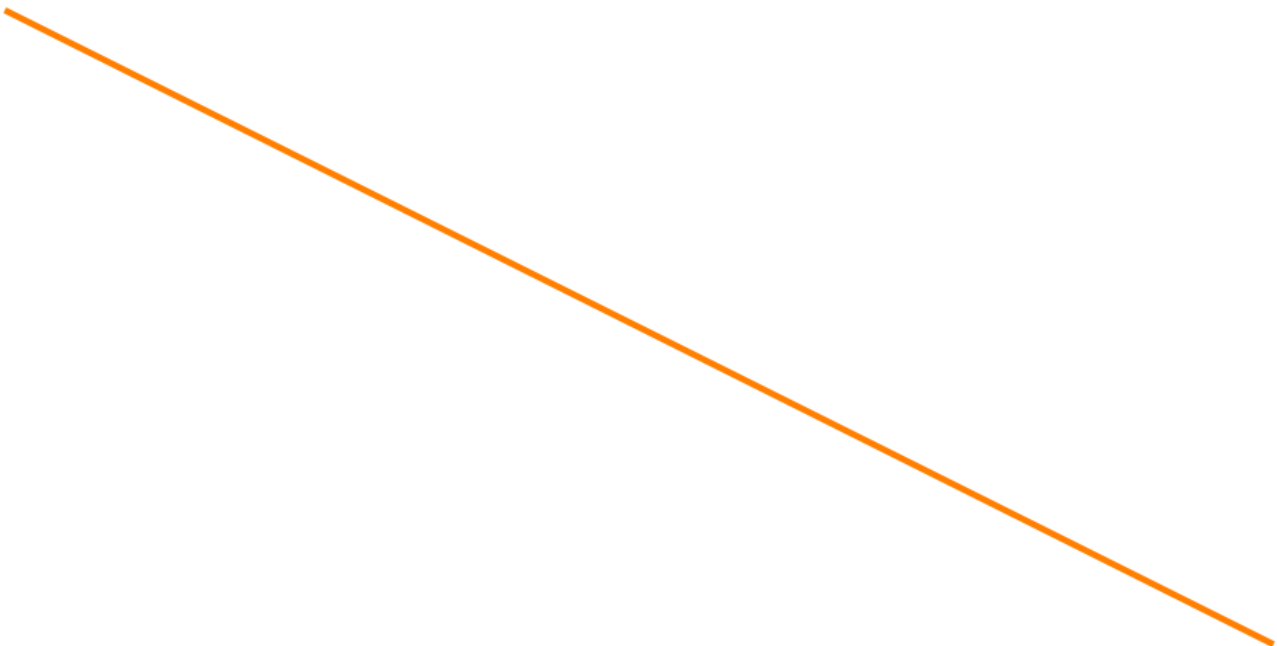
The following code snippets show examples of rendering various shapes.

5.2.1 Lines

Using `line()` to 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")
```



Drawing 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")
```



5.2.2 Circle

Using `circle()` to draw a disc filled in pink with a grey outline:

```
from fpdf import FPDF

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



5.2.3 Ellipse

Using `ellipse()`, 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=100, h=50, style="FD")
pdf.output("ellipse.pdf")
```



5.2.4 Rectangle

Using `rect()` to 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")
```



Using `rect()` to draw rectangles with round corners:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_draw_color(200)
y = 10
pdf.rect(60, y, 33, 28, round_corners=True, style="D")

pdf.set_fill_color(0, 255, 0)
pdf.rect(100, y, 50, 10, round_corners=("BOTTOM_RIGHT"), style="DF")

pdf.set_fill_color(255, 255, 0)
pdf.rect(160, y, 10, 10, round_corners=("TOP_LEFT", "BOTTOM_LEFT"), style="F")
pdf.output("round_corners_rectangles.pdf")
```



5.2.5 Polygon

Using `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, style="DF")
pdf.output("polygon.pdf")
```



5.2.6 Arc

Using `arc()` :

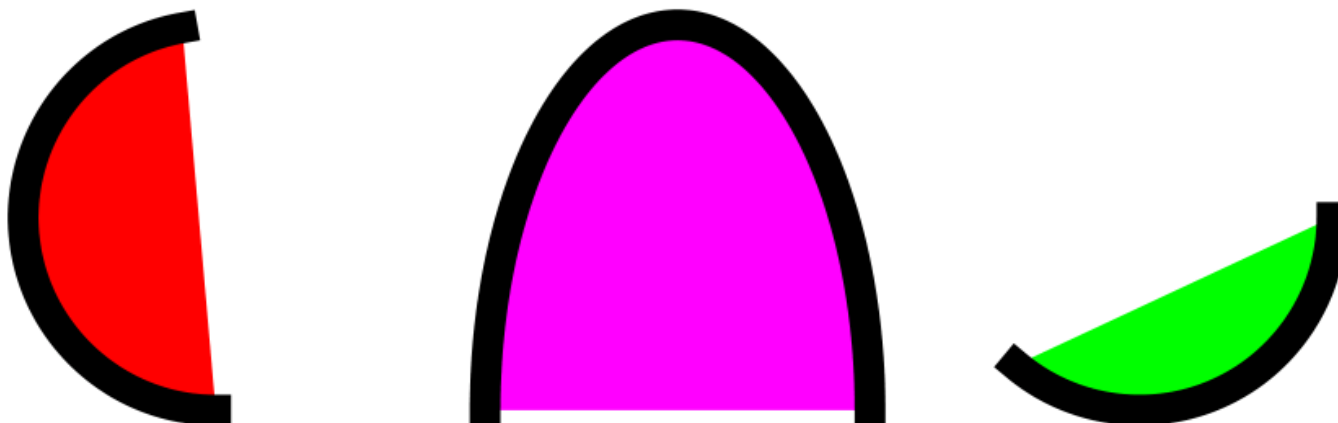
```
from fpdf import FPDF

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=90, end_angle=260, style="FD")

pdf.set_fill_color(r=255, g=0, b=255)
pdf.arc(x=105, y=75, a=25, b=50, start_angle=180, end_angle=360, style="FD")

pdf.set_fill_color(r=0, g=255, b=0)
pdf.arc(x=135, y=75, a=25, b=25, start_angle=0, end_angle=130, style="FD")

pdf.output("arc.pdf")
```

5.2.7 Solid arc

Using `solid_arc()` :

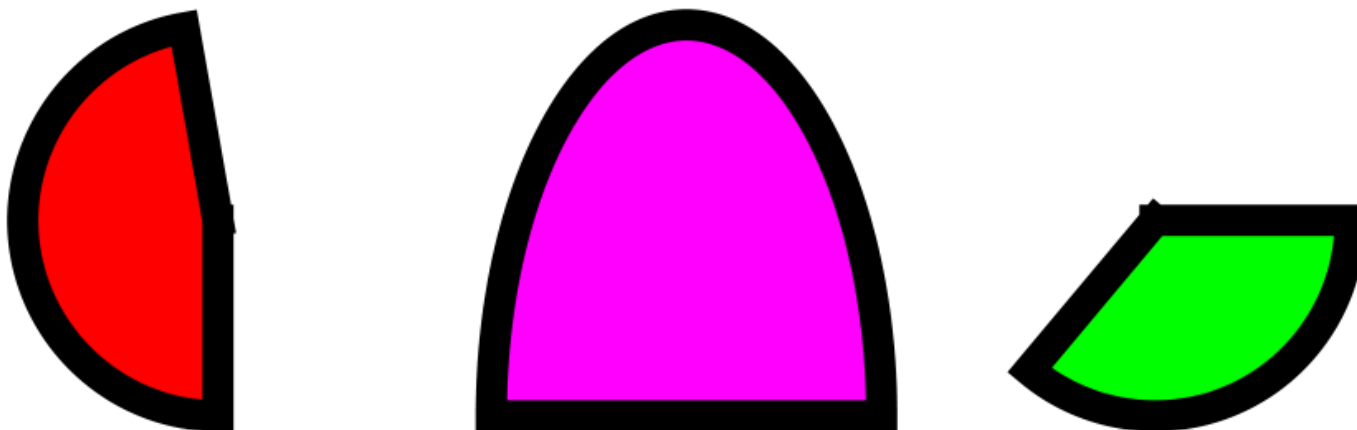
```
from fpdf import FPDF

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=90, end_angle=260, style="FD")

pdf.set_fill_color(r=255, g=0, b=255)
pdf.solid_arc(x=105, y=75, a=25, b=50, start_angle=180, end_angle=360, style="FD")

pdf.set_fill_color(r=0, g=255, b=0)
pdf.solid_arc(x=135, y=75, a=25, b=25, start_angle=0, end_angle=130, style="FD")

pdf.output("solid_arc.pdf")
```



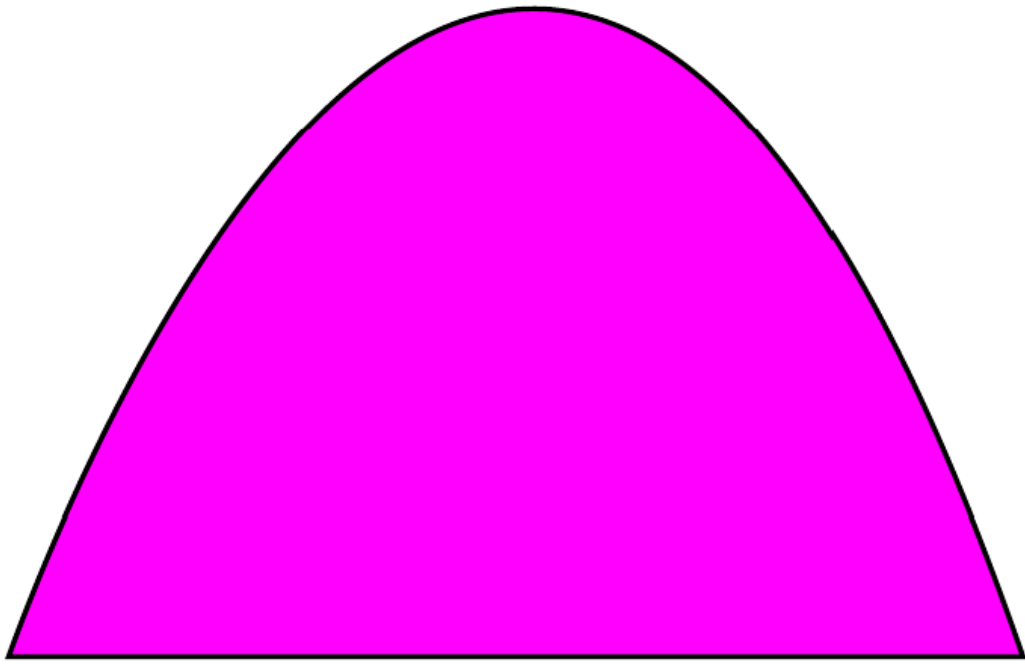
5.2.8 Bezier Curve

New in  2.7.10

Using `bezier()` to create a cubic Bézier curve:

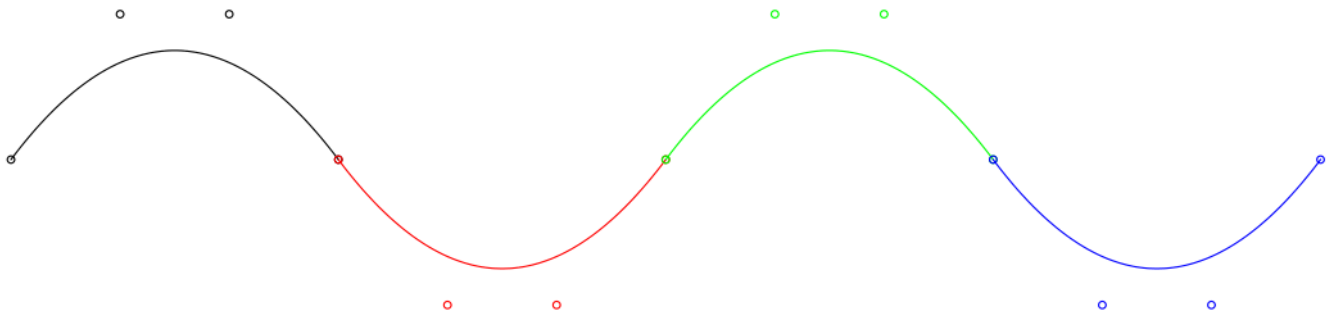
```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_fill_color(r=255, g=0, b=255)
pdf.bezier([(20, 80), (40, 20), (60, 80)], style="DF")
pdf.output("bezier.pdf")
```

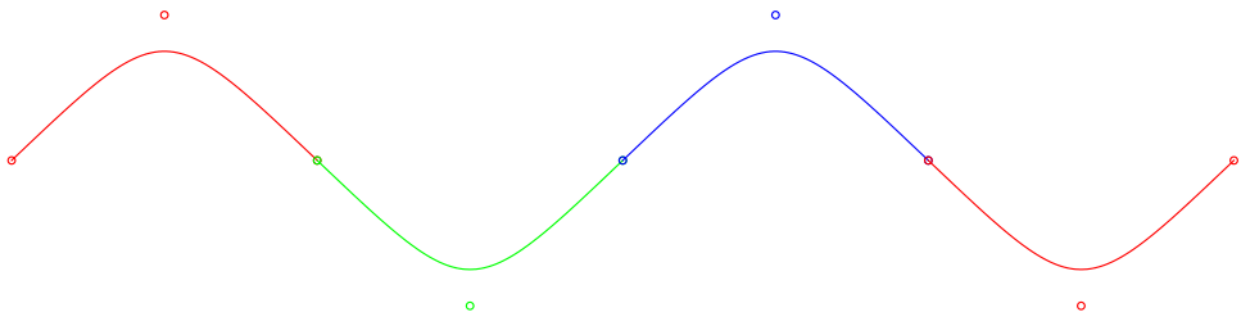


One of the nice properties of Bézier curves is that they can be chained:

Chaining quadratic curves:



Chaining cubic curves:



Note that, for smooth joining cubic Bézier curves, neighbor control points around the joining point must mirror each other (*cf.* [Wikipedia](#)).

Source code: `test_bezier_chaining()` in `test_bezier.py`

5.2.9 Regular Polygon

Using `regular_polygon()` :

```
from fpdf import FPDF

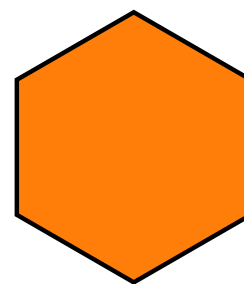
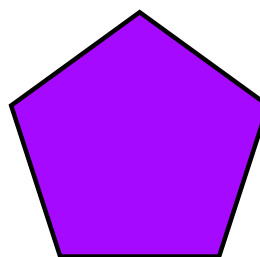
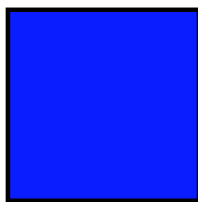
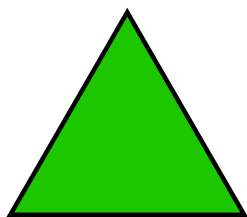
pdf = FPDF()
pdf.add_page()
pdf.set_line_width(0.5)

pdf.set_fill_color(r=30, g=200, b=0)
pdf.regular_polygon(x=40, y=80, polyWidth=30, rotateDegrees=270, numSides=3, style="FD")

pdf.set_fill_color(r=10, g=30, b=255)
pdf.regular_polygon(x=80, y=80, polyWidth=30, rotateDegrees=135, numSides=4, style="FD")

pdf.set_fill_color(r=165, g=10, b=255)
pdf.regular_polygon(x=120, y=80, polyWidth=30, rotateDegrees=198, numSides=5, style="FD")

pdf.set_fill_color(r=255, g=125, b=10)
pdf.regular_polygon(x=160, y=80, polyWidth=30, rotateDegrees=270, numSides=6, style="FD")
pdf.output("regular_polygon.pdf")
```



5.2.10 Regular Star

Using `star()`:

```
from fpdf import FPDF

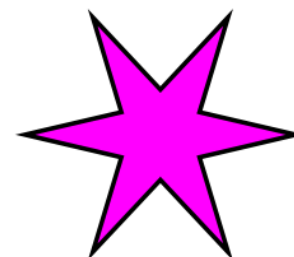
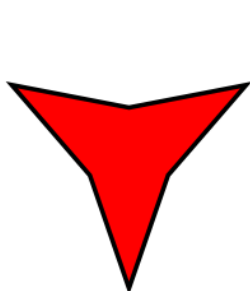
pdf = FPDF()
pdf.add_page()
pdf.set_line_width(0.5)

pdf.set_fill_color(r=255, g=0, b=0)
pdf.star(x=40, y=80, r_in=5, r_out=15, rotate_degrees=0, corners=3, style="FD")

pdf.set_fill_color(r=0, g=255, b=255)
pdf.star(x=80, y=80, r_in=5, r_out=15, rotate_degrees=90, corners=4, style="FD")

pdf.set_fill_color(r=255, g=255, b=0)
pdf.star(x=120, y=80, r_in=5, r_out=15, rotate_degrees=180, corners=5, style="FD")

pdf.set_fill_color(r=255, g=0, b=255)
pdf.star(x=160, y=80, r_in=5, r_out=15, rotate_degrees=270, corners=6, style="FD")
pdf.output("star.pdf")
```

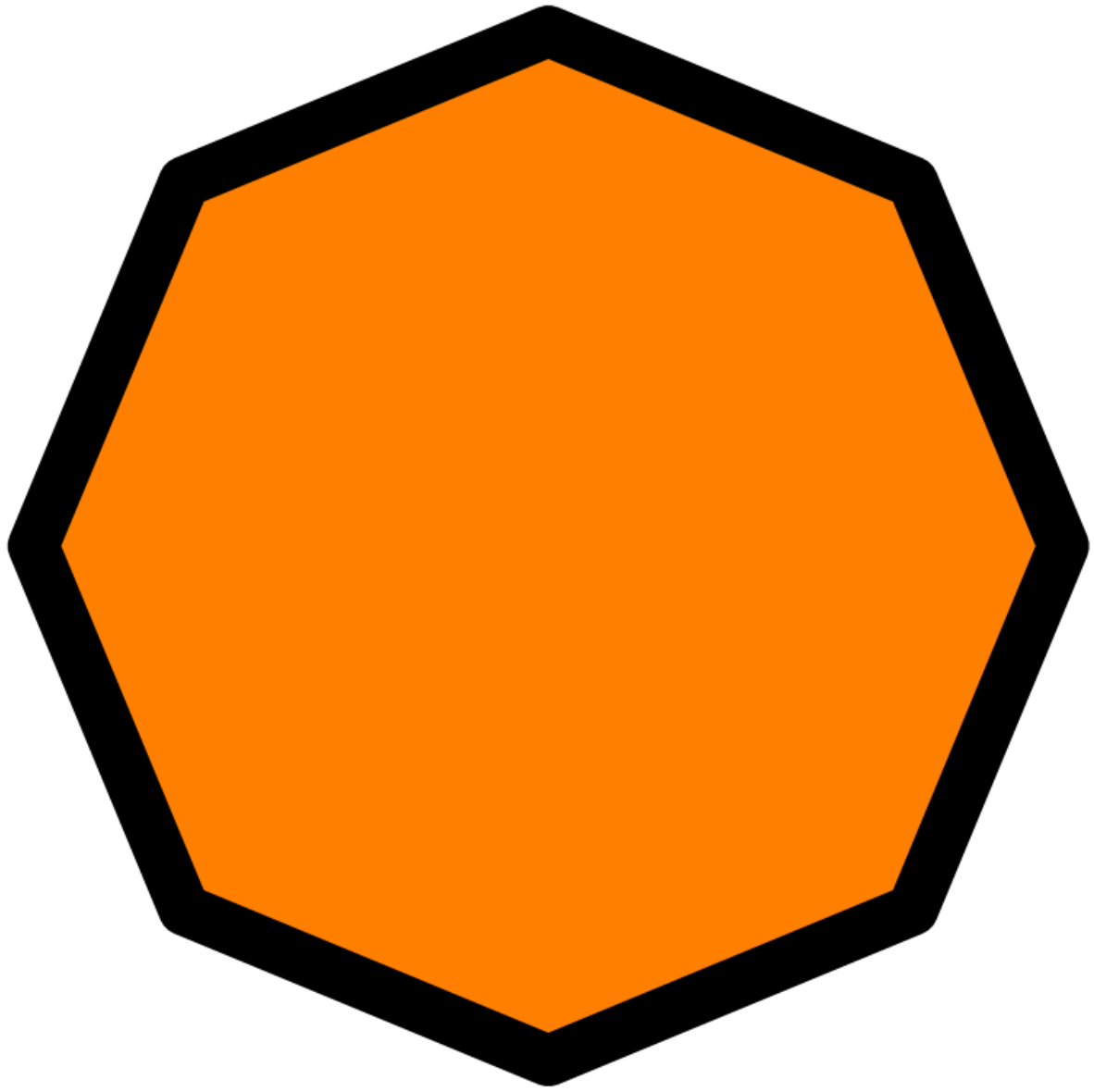


5.2.11 Path styling

- `line_width` specifies the thickness of the line used to stroke a path
- `stroke_join_style` defines how the corner joining two path components should be rendered:

```
from fpdf import FPDF
from fpdf.enums import StrokeJoinStyle

pdf = FPDF()
pdf.add_page()
pdf.set_line_width(5)
pdf.set_fill_color(r=255, g=128, b=0)
with pdf.local_context(stroke_join_style=StrokeJoinStyle.ROUND):
    pdf.regular_polygon(x=50, y=120, polyWidth=100, numSides=8, style="FD")
pdf.output("regular_polygon_rounded.pdf")
```



- `stroke_cap_style` defines how the end of a stroke should be rendered. This affects the ends of the segments of dashed strokes, as well.

```
from fpdf import FPDF
from fpdf.enums import StrokeCapStyle

pdf = FPDF()
pdf.add_page()
pdf.set_line_width(5)
pdf.set_fill_color(r=255, g=128, b=0)
with pdf.local_context(stroke_cap_style=StrokeCapStyle.ROUND):
    pdf.line(x1=50, y1=50, x2=150, y2=100)
pdf.output("line_with_round_ends.pdf")
```

There are even more specific path styling settings supported: `dash_pattern`, `stroke_opacity`, `stroke_miter_limit`...

All of those settings can be set in a `local_context()`.

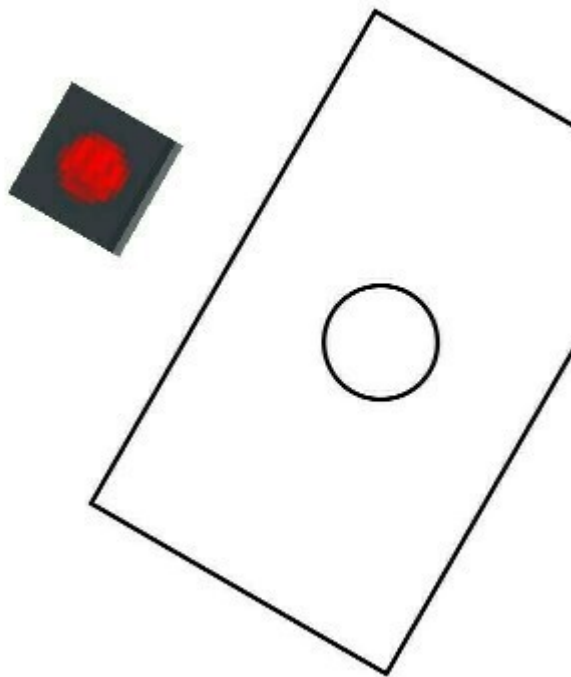
5.3 Transformations

5.3.1 Rotation

The `rotation()` context-manager will apply a rotation to all objects inserted in its indented block:

```
from fpdf import FPDF

pdf = FPDF(format=(40, 40))
pdf.add_page()
x, y = 15, 15
with pdf.rotation(60, x=x, y=y):
    pdf.circle(x=x, y=y+15, r=5)
    # Inserting a small base64-encoded image:
    pdf.image("data:image/
png;base64,iVBORw0KGgoAAAANSUgAAABAAAAQBAMAAADt3eJSAAAMFBMVEU00kArMjhobHEoPUPFEIu00L+AAC2FBZ2JyuNICOfGx7xAwTjCA1CNTvVDA1aLzQ3C0jMAAAAVU1EQVQI12NgwAaCDSA08
x=x, y=y)
    pdf.rect(x=x-10, y=y+10, w=25, h=15)
pdf.output("rotations.pdf")
```



5.3.2 Skew

`skew` creates a skewing transformation of magnitude `ax` in the horizontal axis and `ay` in the vertical axis. The transformation originates from `x`, `y` and will use a default origin unless specified otherwise:

```
with pdf.skew(ax=0, ay=10):
    pdf.cell(text="text skewed on the y-axis")
```

text skewed on the y-axis

```
with pdf.skew(ax=10, ay=0):
    pdf.cell(text="text skewed on the x-axis")
```

text skewed on the x-axis

```
pdf.set_line_width(2)
pdf.set_draw_color(240)
pdf.set_fill_color(r=230, g=30, b=180)
with pdf.skew(ax=-45, ay=0, x=100, y=170):
    pdf.circle(x=100, y=170, r=10, style="FD")
```



5.3.3 Mirror

New in  2.7.5

The `mirror` context-manager applies a mirror transformation to all objects inserted in its indented block over a given mirror line by specifying starting co-ordinate and angle.

```
x, y = 100, 100
pdf.text(x, y, text="mirror this text")
with pdf.mirror((x, y), "EAST"):
    pdf.set_text_color(r=255, g=128, b=0)
    pdf.text(x, y, text="mirror this text")
```

mirror this text
mirror this text

```
pdf.text(x, y, text="mirror this text")
with pdf.mirror((x, y), "NORTH"):
    pdf.set_text_color(r=255, g=128, b=0)
    pdf.text(x, y, text="mirror this text")
```

txet zint rorrimmirror this text

```
prev_x, prev_y = pdf.x, pdf.y
pdf.multi_cell(w=50, text=LOREM_IPSUM)
with pdf.mirror((pdf.x, pdf.y), "NORTHEAST"):
    # Reset cursor to mirror original multi-cell
    pdf.x = prev_x
    pdf.y = prev_y
    pdf.multi_cell(w=50, text=LOREM_IPSUM, fill=True)
```

Lorem ipsum Ut nostrud
irure reprehenderit anim
nostrud dolore sed ut
Excepteur dolore ut sunt
irure

irure Excepteur dolore ut sunt
nostrud dolore sed ut
irure reprehenderit anim
Ut nostrud dolore sed ut
Excepteur dolore ut sunt

5.4 Transparency

The alpha opacity of [text](#), [shapes](#) and even [images](#) can be controlled through `stroke_opacity` (for lines) & `fill_opacity` (for all other content types):

```
pdf = FPDF()
pdf.set_font("Helvetica", "B", 24)
pdf.set_line_width(1.5)
pdf.add_page()

# Draw an opaque red square:
pdf.set_fill_color(255, 0, 0)
pdf.rect(10, 10, 40, 40, "DF")

# Set alpha to semi-transparency for shape lines & filled areas:
with pdf.local_context(fill_opacity=0.5, stroke_opacity=0.5):
    # Draw a green square:
    pdf.set_fill_color(0, 255, 0)
    pdf.rect(20, 20, 40, 40, "DF")

# Set transparency for images & text:
with pdf.local_context(fill_opacity=0.25):
    # Insert an image:
    pdf.image(HERE / "../docs/fpdf2-logo.png", 30, 30, 40)
    # Print some text:
    pdf.text(22, 29, "You are...")

# Print some text with full opacity:
pdf.text(30, 45, "Over the top")

# Produce the resulting PDF:
pdf.output("transparency.pdf")
```

Results in:



5.5 Barcodes

5.5.1 Code 39

Here is an example on how to generate a [Code 39](#) barcode:

```
from fpdf import FPDF

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

Output preview:



5.5.2 Interleaved 2 of 5

Here is an example on how to generate an [Interleaved 2 of 5](#) barcode:

```
from fpdf import FPDF

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

Output preview:



5.5.3 PDF-417

Here is an example on how to generate a [PDF-417](#) barcode using the [pdf417](#) lib:

```
from fpdf import FPDF
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:



5.5.4 QRCode

Here is an example on how to generate a [QR Code](#) using the [python-qrcode](#) lib:

```
from fpdf import FPDF
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:



5.5.5 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.

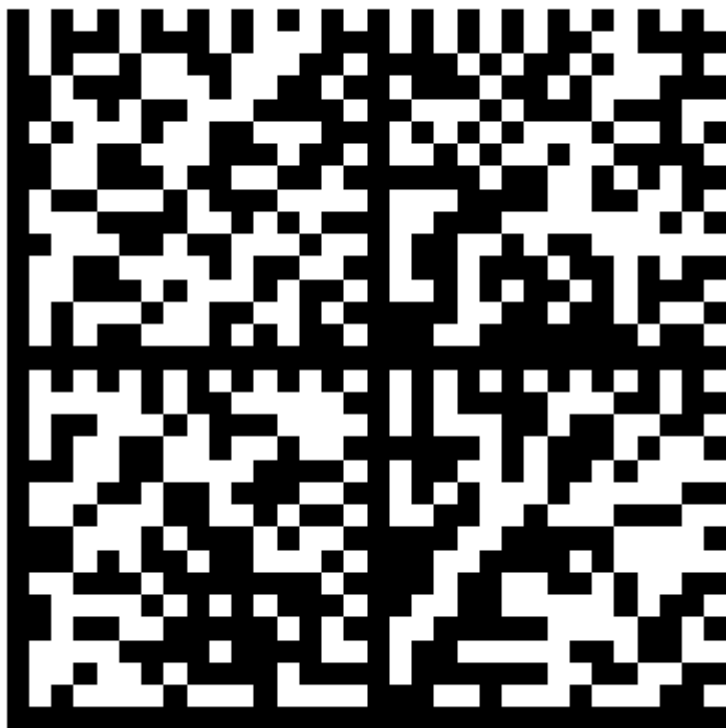
```
from fpdf import FPDF
from pystrich.datamatrix import DataMatrixEncoder, DataMatrixRenderer

# 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)
pdf.output("datamatrix.pdf")
```



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

class PDF(FPDF):
    def datamatrix(self, text, w, h=None, x=None, y=None, cellsize=5):
```

```

        if x is None:
            x = self.x
        if y is None:
            y = self.y
        if h is None:
            h = w
        encoder = DataMatrixEncoder(text)
        encoder.width = w
        encoder.height = h
        renderer = DataMatrixRenderer(encoder.matrix, encoder.regions)
        img = renderer.get_pilimage(celldsize)
        self.image(img, x, y, w, h)

# Usage example:
pdf = PDF()
pdf.add_page()
pdf.set_font("Helvetica", size=24)
pdf.datamatrix("Hello world!", w=100)
pdf.output("datamatrix_from_method.pdf")

```

5.5.6 Code128

Here is an example on how to generate a [Code 128](#) barcode using the [python-barcode](#) lib, that can be installed by running `pip install python-barcode`:

```

from io import BytesIO
from fpdf import FPDF
from barcode import Code128
from barcode.writer import SVGWriter

# Create a new PDF document
pdf = FPDF()
pdf.add_page()

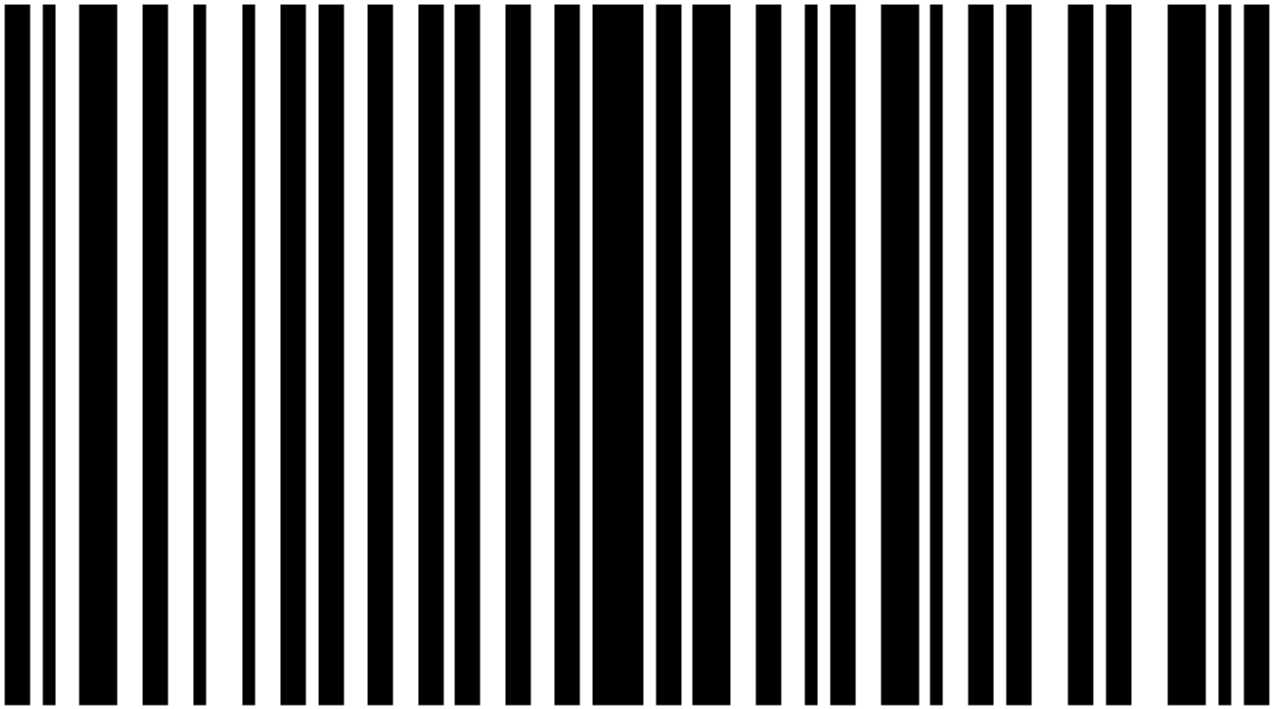
# Set the position and size of the image in the PDF
x = 50
y = 50
w = 100
h = 70

# Generate a Code128 Barcode as SVG:
svg_img_bytes = BytesIO()
Code128("100000902922", writer=SVGWriter()).write(svg_img_bytes)
pdf.image(svg_img_bytes, x=x, y=y, w=w, h=h)

# Output a PDF file:
pdf.output("code128_barcode.pdf")

```

Output Preview:



5.6 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).

5.6.1 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.

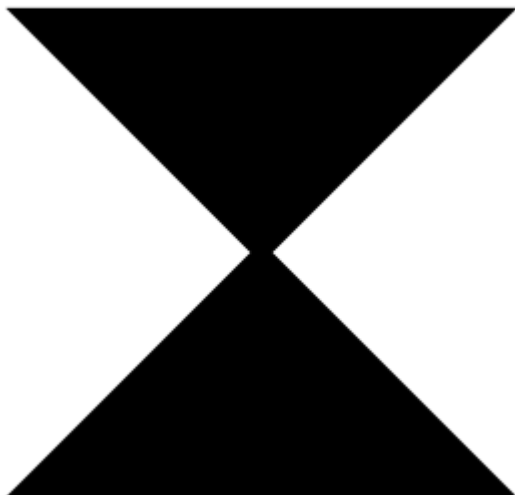
```
import fpdf

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](#)

5.6.2 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

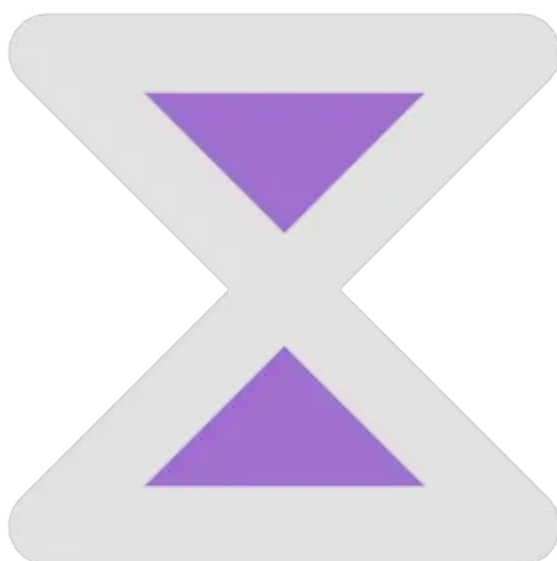
pdf = fpdf.FPDF(unit='mm', format=(10, 10))
pdf.add_page()

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](#)

5.6.3 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 created 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

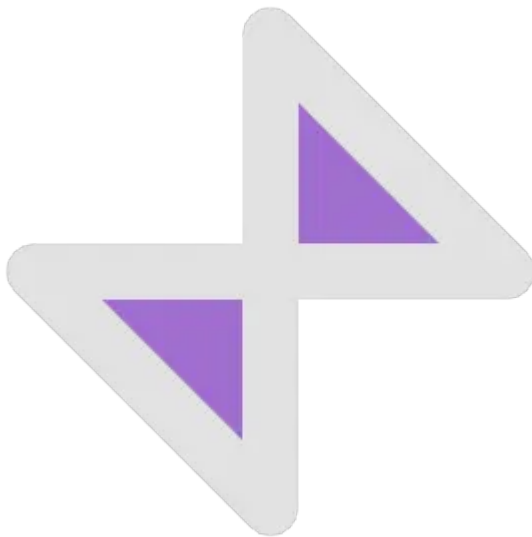
pdf = fpdf.FPDF(unit="mm", format=(10, 10))
pdf.add_page()

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.

5.6.4 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](#)

5.6.5 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.

5.7 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.

5.7.1 Basic usage

SVG files can be directly inserted inside a PDF file using the `image()` method:

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.image("vector.svg")
pdf.output("doc-with-svg.pdf")
```

Either the embedded `.svg` file must include `width` and/or `height` attributes (absolute or relative), or some dimensions must be provided to `.image()` through its `w=` and/or `h=` parameters.

5.7.2 Detailed example

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

```
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")
```

5.7.3 Converting vector graphics to raster graphics

Usually, embedding SVG as vector graphics in PDF documents is the best approach, as it is both lightweight and will allow for better details / precision of the images inserted.

But sometimes, SVG images cannot be directly embedded as vector graphics (SVG), and a conversion to raster graphics (PNG, JPG) must be performed.

The following sections demonstrate how to perform such conversion, using [Pygal charts](#) as examples:

Using cairosvg

A faster and efficient approach for embedding [Pygal](#) SVG charts into a PDF file is to use the [cairosvg](#) library to convert the vector graphics generated into a [BytesIO](#) instance, so that we can keep these data in an in-memory buffer:

```
import pygal
from fpdf import FPDF
from io import BytesIO
import cairosvg

# Create a Pygal bar chart
bar_chart = pygal.Bar()
bar_chart.title = 'Browser usage evolution (in %)'
bar_chart.x_labels = map(str, range(2002, 2013))
bar_chart.add('Firefox', [None, None, 0, 16.6, 25, 31, 36.4, 45.5, 46.3, 42.8, 37.1])
bar_chart.add('Chrome', [None, None, None, None, None, None, 0, 3.9, 10.8, 23.8, 35.3])
bar_chart.add('IE', [85.8, 84.6, 84.7, 74.5, 66, 58.6, 54.7, 44.8, 36.2, 26.6, 20.1])
bar_chart.add('Others', [14.2, 15.4, 15.3, 8.9, 9, 10.4, 8.9, 5.8, 6.7, 6.8, 7.5])
svg_img = bar_chart.render()

# Convert the SVG chart to a PNG image in a BytesIO object
img_bytesio = BytesIO()
cairosvg.svg2png(svg_img, write_to=img_bytesio, dpi=96)

# Set the position and size of the image in the PDF
x = 50
y = 50
w = 100
h = 70

# Build the PDF
pdf = FPDF()
pdf.add_page()
pdf.image(img_bytesio, x=x, y=y, w=w, h=h)
pdf.output('browser-usage-bar-chart.pdf')
```

The above code generates a PDF with the following graph:



!! Troubleshooting advice !!

You may encounter `GTK` (Gnome Toolkit) errors while executing the above example in windows. Error could be like following -

```

OSError: no library called "cairo-2" was found
no library called "cairo" was found
no library called "libcairo-2" was found
cannot load library 'libcairo.so.2': error 0x7e
cannot load library 'libcairo.2.dylib': error 0x7e
cannot load library 'libcairo-2.dll': error 0x7e

```

In this case install `GTK` from [GTK-for-Windows-Runtime-Environment-Installer](#). Restart your editor. And you are all done.

Using `svglib` and `reportlab`

An alternative, purely pythonic but slightly slower solution is to use `reportlab` and `svglib`:

```

import io
import pygal
from reportlab.graphics import renderPM
from svglib.svglib import SvgRenderer
from fpdf import FPDF
from lxml import etree

# Create a Pygal bar chart
bar_chart = pygal.Bar()
bar_chart.title = 'Sales by Year'
bar_chart.x_labels = ['2016', '2017', '2018', '2019', '2020']
bar_chart.add('Product A', [500, 750, 1000, 1250, 1500])
bar_chart.add('Product B', [750, 1000, 1250, 1500, 1750])
svg_img = bar_chart.render()

# Convert the SVG chart to a JPEG image in a BytesIO object
drawing = SvgRenderer('').render(etree.fromstring(svg_img))

```

```

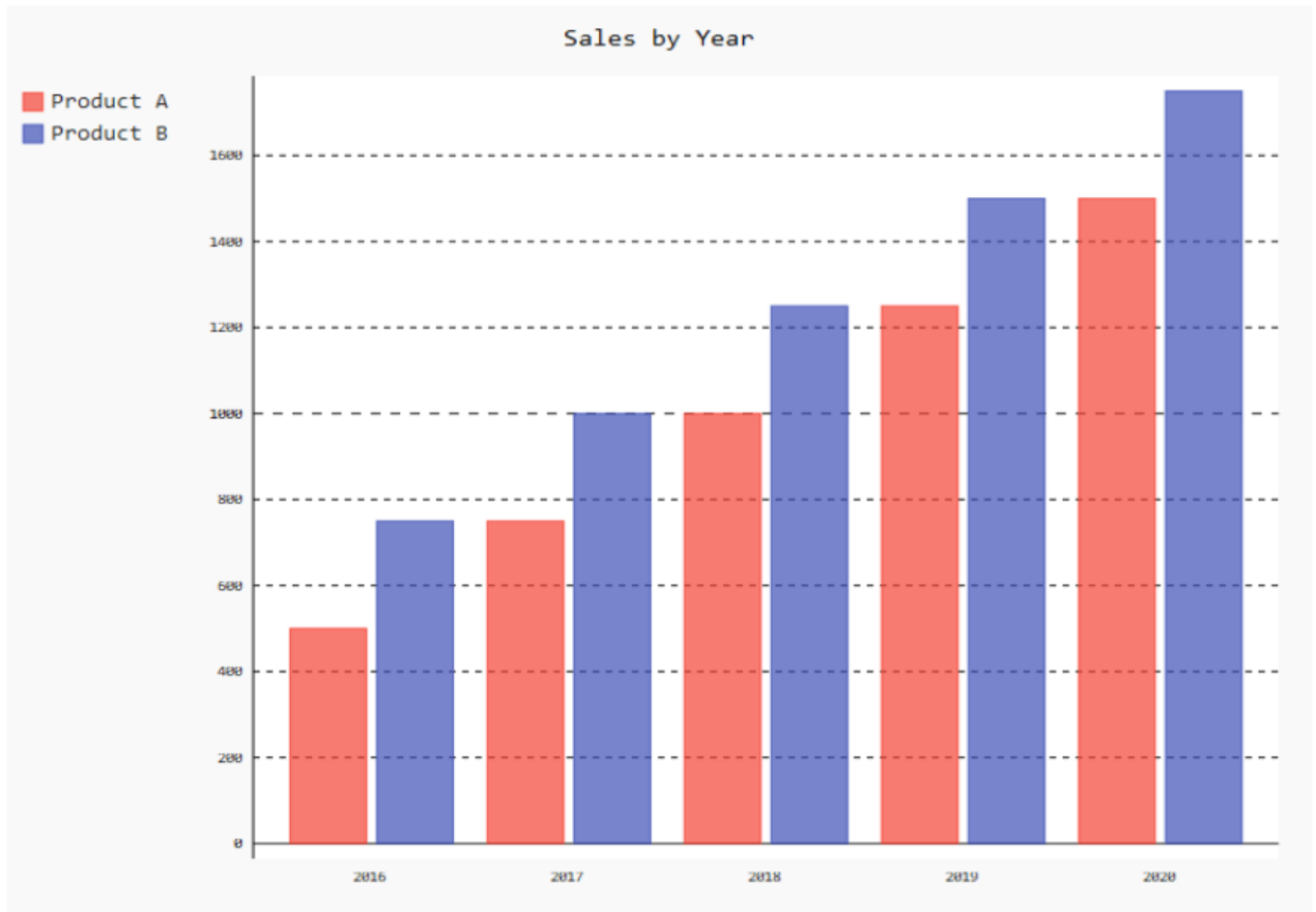
jpg_img_bytes = renderPM.drawToString(drawing, fmt='JPG', dpi=72)
img_bytesio = io.BytesIO(jpg_img_bytes)

# Set the position and size of the image in the PDF
x = 50
y = 50
w = 100
h = 70

# Build the PDF
pdf = FPDF()
pdf.add_page()
pdf.image(img_bytesio, x=x, y=y, w=w, h=h)
pdf.output('sales-by-year-bar-chart.pdf')

```

The above code generates the following output:



Performance considerations

Regarding performance, `cairosvg` is generally faster than `svglib` when it comes to rendering SVG files to other formats. This is because `cairosvg` is built on top of a fast C-based rendering engine, while `svglib` is written entirely in Python, and hence a bit slower. Additionally, `cairosvg` offers various options for optimizing the rendering performance, such as disabling certain features, like fonts or filters.

5.7.4 Warning logs

The `fpdf.svg` module produces `WARNING` log messages for **unsupported** SVG tags & attributes. If need be, you can suppress those logs:

```

logging.getLogger("fpdf.svg").propagate = False

```


5.7.5 Supported SVG Features

- groups (`<g>`)
- paths (`<path>`)
- basic shapes (`<rect>`, `<circle>`, `<ellipse>`, `<line>`, `<polyline>`, `<polygon>`)
- basic `<image>` elements
- basic cross-references, with `defs` tags anywhere in the SVG code
- stroke & fill coloring and opacity
- basic stroke styling
- inline CSS styling via `style="..."` attributes
- clipping paths

5.7.6 Currently Unsupported Notable SVG Features

Everything not listed as supported is unsupported, which is a lot. SVG is a very 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.

There are some common SVG features that are currently **unsupported**, but that `fpdf2` could end up supporting with the help of contributors :

- `<tspan>` / `<textPath>` / `<text>` (-> there is a starting [draft PR](#))
- `<symbol>`
- `<marker>`
- `<pattern>`
- gradients: `<linearGradient>` & `<radialGradient>`
- embedded non-image content (including nested SVGs)
- many standard attributes
- CSS styling via `<style>` tags or external *.css files.

Contributions would be very welcome to add support for more SVG features! 👍

If you are interested in contributing to `fpdf2` regarding this, drop a comment on GitHub issue [#537](#) and a maintainer will give some pointers to start poking with the code 😊

5.8 Charts & graphs

5.8.1 Charts

Using Matplotlib

Before running this example, please install the required dependencies using the command below:

```
pip install fpdf2 matplotlib
```

Example taken from [Matplotlib artist tutorial](#):

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

fig = Figure(figsize=(6, 4), dpi=300)
fig.subplots_adjust(top=0.8)
ax1 = fig.add_subplot(211)
ax1.set_ylabel("volts")
ax1.set_title("a sine wave")

t = np.arange(0.0, 1.0, 0.01)
s = np.sin(2 * np.pi * t)
(line,) = ax1.plot(t, s, color="blue", lw=2)

# Fixing random state for reproducibility
np.random.seed(19680801)

ax2 = fig.add_axes([0.15, 0.1, 0.7, 0.3])
n, bins, patches = ax2.hist(
    np.random.randn(1000), 50, facecolor="yellow", edgecolor="yellow"
)
ax2.set_xlabel("time (s)")

# Converting Figure to an image:
canvas = FigureCanvas(fig)
canvas.draw()
img = Image.fromarray(np.asarray(canvas.buffer_rgba()))

pdf = FPDF()
pdf.add_page()
pdf.image(img, w=pdf.epw) # Make the image full width
pdf.output("matplotlib.pdf")
```

Result:



You can also embed a figure as [SVG](#):

```
from fpdf import FPDF
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=[2, 2])
x = np.arange(0, 10, 0.00001)
y = x*np.sin(2* np.pi * x)
plt.plot(y)
plt.savefig("figure.svg", format="svg")

pdf = FPDF()
pdf.add_page()
pdf.image("figure.svg")
pdf.output("doc-with-figure.pdf")
```

Using Pandas

The dependencies required for the following examples can be installed using this command:

```
pip install fpdf2 matplotlib pandas
```

Create a plot using `pandas.DataFrame.plot`:

```
from io import BytesIO
from fpdf import FPDF
import pandas as pd
import matplotlib.pyplot as plt
import io

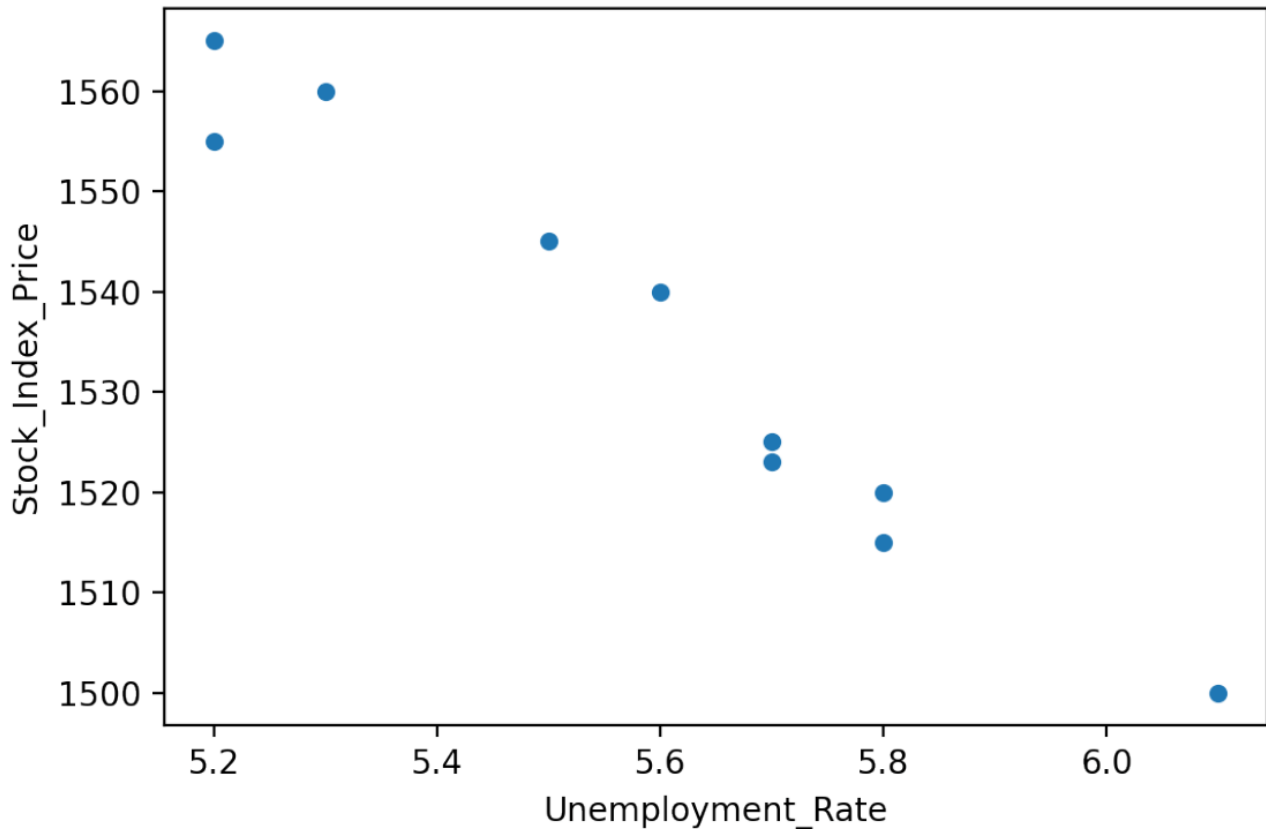
DATA = {
    "Unemployment_Rate": [6.1, 5.8, 5.7, 5.7, 5.8, 5.6, 5.5, 5.3, 5.2, 5.2],
    "Stock_Index_Price": [1500, 1520, 1525, 1523, 1515, 1540, 1545, 1560, 1555, 1565],
}
COLUMNS = tuple(DATA.keys())

plt.figure() # Create a new figure object
df = pd.DataFrame(DATA, columns=COLUMNS)
df.plot(x=COLUMNS[0], y=COLUMNS[1], kind="scatter")
```

```
# Converting Figure to an image:
img_buf = BytesIO() # Create image object
plt.savefig(img_buf, dpi=200) # Save the image

pdf = FPDF()
pdf.add_page()
pdf.image(img_buf, w=pdf.epw) # Make the image full width
pdf.output("matplotlib_pandas.pdf")
img_buf.close()
```

Result:



Create a table with pandas [DataFrame](#):

```
from fpdf import FPDF
import pandas as pd

DF = pd.DataFrame(
    {
        "First name": ["Jules", "Mary", "Carlson", "Lucas"],
        "Last name": ["Smith", "Ramos", "Banks", "Cimon"],
        "Age": [34, 45, 19, 31],
        "City": ["San Juan", "Orlando", "Los Angeles", "Saint-Mahturin-sur-Loire"],
    }
)
# Convert all data inside dataframe into string type:
).applymap(str)

COLUMNS = [list(DF)] # Get list of dataframe columns
ROWS = DF.values.tolist() # Get list of dataframe rows
DATA = COLUMNS + ROWS # Combine columns and rows in one list

pdf = FPDF()
pdf.add_page()
pdf.set_font("Times", size=10)
with pdf.table(
    borders_layout="MINIMAL",
    cell_fill_color=200, # grey
    cell_fill_mode="ROWS",
    line_height=pdf.font_size * 2.5,
    text_align="CENTER",
    width=160,
) as table:
    for data_row in DATA:
        row = table.row()
        for datum in data_row:
```

```
row.cell(datum)
pdf.output("table_from_pandas.pdf")
```

Result:

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

Using Plotly

Before running this example, please install the required dependencies using the command below:

```
pip install fpdf2 plotly kaleido numpy
```

[kaleido](#) is a cross-platform library for generating static images that is used by plotly.

Example taken from [Plotly static image export tutorial](#):

```
import io
import plotly.graph_objects as go
import numpy as np
from fpdf import FPDF

np.random.seed(1)

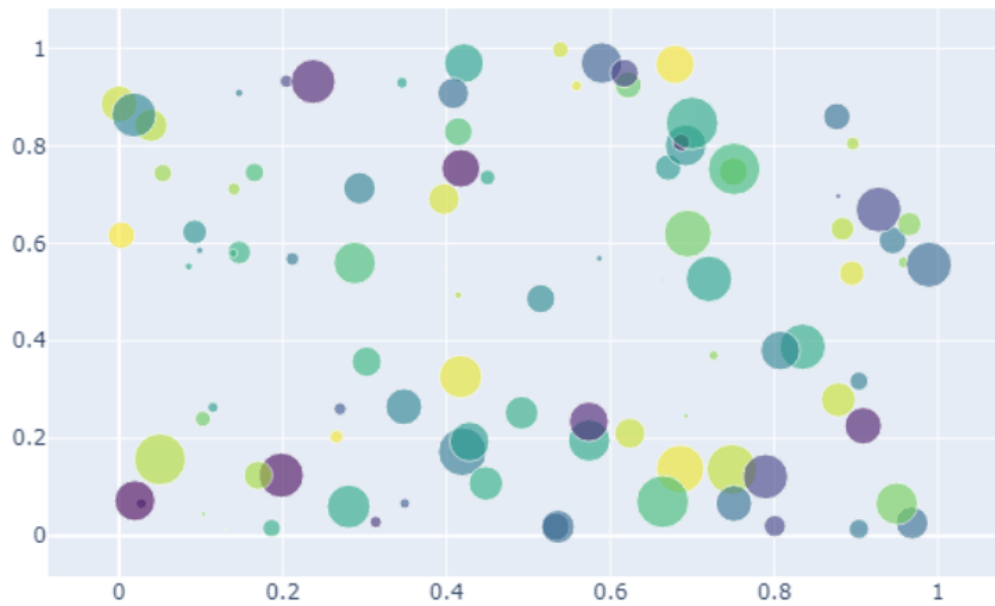
N = 100
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
sz = np.random.rand(N) * 30

fig = go.Figure()
fig.add_trace(
    go.Scatter(
        x=x,
        y=y,
        mode="markers",
        marker=go.scatter.Marker(
            size=sz, color=colors, opacity=0.6, colorscale="Viridis"
        ),
    )
)

# Convert the figure to png using kaleido
image_data = fig.to_image(format="png", engine="kaleido")

# Create an io.BytesIO object which can be used by FPDF2
image = io.BytesIO(image_data)
pdf = FPDF()
pdf.add_page()
pdf.image(image, w=pdf.epw) # Width of the image is equal to the width of the page
pdf.output("plotly_demo.pdf")
```

Result:



You can also embed a figure as [SVG](#) but this is not recommended because the text data such as the x and y axis bars might not show as illustrated in the result image because plotly places this data in a svg text tag which is currently [not supported](#) by FPDF2.

Before running this example, please install the required dependencies:

```
pip install fpdf2 plotly kaleido pandas
```

```
from fpdf import FPDF
import plotly.express as px

fig = px.bar(x=["a", "b", "c"], y=[1, 3, 2])
fig.write_image("figure.svg")

pdf = FPDF()
pdf.add_page()
pdf.image("figure.svg", w=pdf.epw)
pdf.output("plotly.pdf")
```

Result:



Using Pygal

[Pygal](#) is a Python graph plotting library. You can install it using: `pip install pygal`

`fpdf2` can embed graphs and charts generated using `Pygal` library. However, they cannot be embedded as SVG directly, because `Pygal` inserts `<style>` & `<script>` tags in the images it produces (cf. [pygal/svg.py](#)), which is currently not supported by `fpdf2`. The full list of supported & unsupported SVG features can be found there: [SVG page](#).

You can find documentation on how to convert vector images (SVG) to raster images (PNG, JPG), with a practical example of embedding PyGal charts, there: [SVG page](#).

5.8.2 Mathematical formulas

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

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: # norec B310
    img = BytesIO(img_file.read())

pdf = FPDF()
pdf.add_page()
pdf.image(img, w=30)
pdf.output("equation_google_charts.pdf")
```

Result:

$$x^n + y^n = a / b$$

Using LaTeX & Matplotlib

Matplotlib can render **LaTeX**: [Text rendering With LaTeX](#).

Example:

```
from io import BytesIO
from fpdf import FPDF
from matplotlib.figure import Figure

fig = Figure(figsize=(6, 2))
gca = fig.gca()
gca.text(0, 0.5, r"$x^n + y^n = \frac{a}{b}$", fontsize=60)
gca.axis("off")

# Converting Figure to a SVG image:
img = BytesIO()
fig.savefig(img, format="svg")

pdf = FPDF()
pdf.add_page()
pdf.image(img, w=100)
pdf.output("equation_matplotlib.pdf")
```

Result:

$$x^n + y^n = \frac{a}{b}$$

If you have trouble with the SVG export, you can also render the matplotlib figure as pixels:

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

fig = Figure(figsize=(6, 2), dpi=300)
gca = fig.gca()
gca.text(0, 0.5, r"$x^n + y^n = \frac{a}{b}$", fontsize=60)
gca.axis("off")

canvas = FigureCanvas(fig)
canvas.draw()
img = Image.fromarray(np.asarray(canvas.buffer_rgba()))

pdf = FPDF()
pdf.add_page()
pdf.image(img, w=100)
pdf.output("equation_matplotlib_raster.pdf")
```


6. PDF Features

6.1 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).

6.1.1 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(text="Cell link", border=1, center=True,
        link="https://github.com/py-pdf/fpdf2")
pdf.output("hyperlink.pdf")
```

6.1.2 Hyperlink with FPDF.multi_cell

```
from fpdf import FPDF

pdf = FPDF()
pdf.set_font("helvetica", size=24)
pdf.add_page()
pdf.multi_cell(
    pdf.epw,
    text="**Website:** [fpdf2](https://py-pdf.github.io/fpdf2/) __Go visit it!__",
    markdown=True,
)
pdf.output("hyperlink.pdf")
```

Links defined this way in Markdown can be styled by setting `FPDF` class attributes `MARKDOWN_LINK_COLOR` (default: `None`) & `MARKDOWN_LINK_UNDERLINE` (default: `True`).

`link="https://...your-url"` can also be used to make the whole cell clickable.

6.1.3 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, text=text)
width = pdf.get_string_width(text)
pdf.link(x=0, y=0, w=width, h=line_height, link="https://github.com/py-pdf/fpdf2")
pdf.output("hyperlink.pdf")
```

6.1.4 Hyperlink with write_html

An alternative method using `FPDF.write_html`:

```
from fpdf import FPDF

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

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

6.1.5 Internal links

Internal links are links redirecting to other pages in the document.

Using `FPDF.cell`:

```
from fpdf import FPDF

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

Other methods can also insert internal links:

- `FPDF.multi_cell` using `link=` **or** `markdown=True` and this syntax: `[link text](page number)`
- `FPDF.link`
- `FPDF.write_html` using anchor tags: `link text`

The unit tests `test_internal_links()` in `test_links.py` provides examples for all of those methods.

6.1.6 Links to other documents on the filesystem

Using `FPDF.cell`:

```
from fpdf import FPDF

pdf = FPDF()
pdf.set_font("helvetica", size=24)
pdf.add_page()
pdf.cell(text="Link to other_doc.pdf", border=1, link="other_doc.pdf")
pdf.output("link_to_other_doc.pdf")
```

Other methods can also insert internal links:

- `FPDF.multi_cell` using `link=` **or** `markdown=True` and this syntax: `[link text](other_doc.pdf)`
- `FPDF.link`
- `FPDF.write_html` using anchor tags: `link text`

The unit test `test_link_to_other_document()` in `test_links.py` provides examples for all of those methods.

6.1.7 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/py-pdf/fpdf2",
        alt_text="GitHub page for fpdf2")
```

6.2 Metadata

The PDF specification contains 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"py-pdf/fpdf{FPDF_VERSION}"
        meta["xmp:CreatorTool"] = __file__
        meta["xmp:MetadataDate"] = datetime.now(datetime.utcnow().astimezone().tzinfo).isoformat()
    pdf.save()
```

6.3 Annotations

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

6.3.1 Text annotations

They are rendered this way by Sumatra PDF reader:



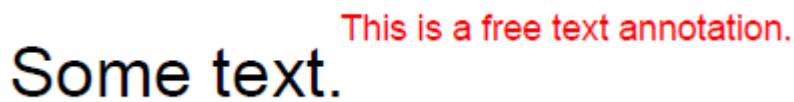
```
from fpdf import FPDF

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

Method documentation: [FPDF.text_annotation](#)

6.3.2 Free Text Annotations

They are rendered this way by Adobe Acrobat Reader:



```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font("Helvetica", size=24)
pdf.text(x=60, y=140, text="Some text.")
pdf.set_draw_color(255, 0, 0)
pdf.set_font_size(12)
pdf.free_text_annotation(
    x=100,
    y=130,
    text="This is a free text annotation.",
    w=150,
    h=15,
)
pdf.output("free_text_annotation.pdf")
```

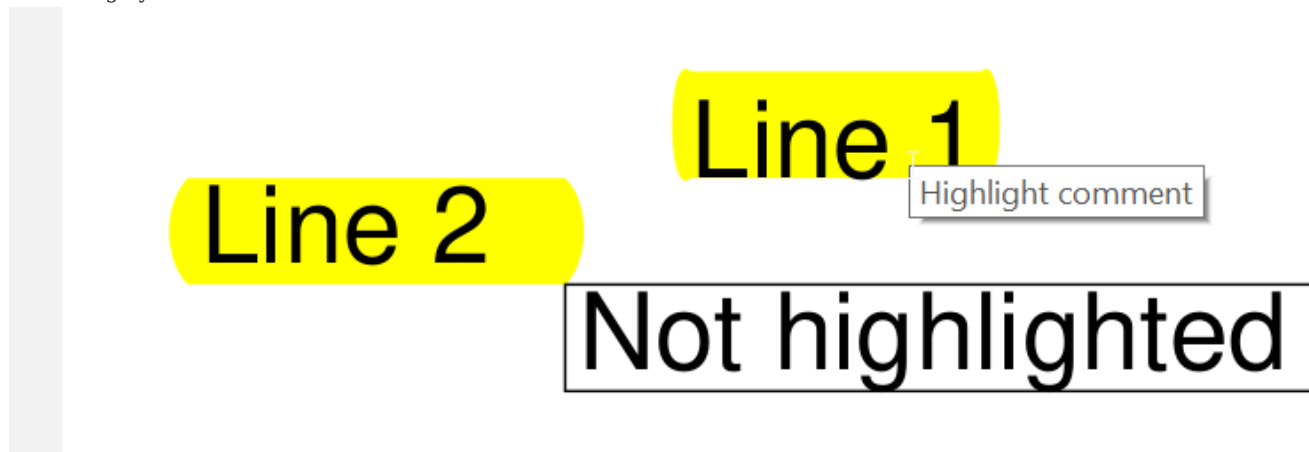
Method documentation: [FPDF.free_text_annotation](#)

6.3.3 Highlights

```
from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.set_font("Helvetica", size=24)
with pdf.highlight("Highlight comment"):
    pdf.text(50, 50, "Line 1")
    pdf.set_y(50)
    pdf.multi_cell(w=30, text="Line 2")
pdf.cell(w=60, text="Not highlighted", border=1)
pdf.output("highlighted.pdf")
```

Rendering by Sumatra PDF reader:



Method documentation: [FPDF.highlight](#)

The appearance of the "highlight effect" can be controlled through the `type` argument: it can be `Highlight` (default), `Underline`, `Squiggly` or `StrikeOut`.

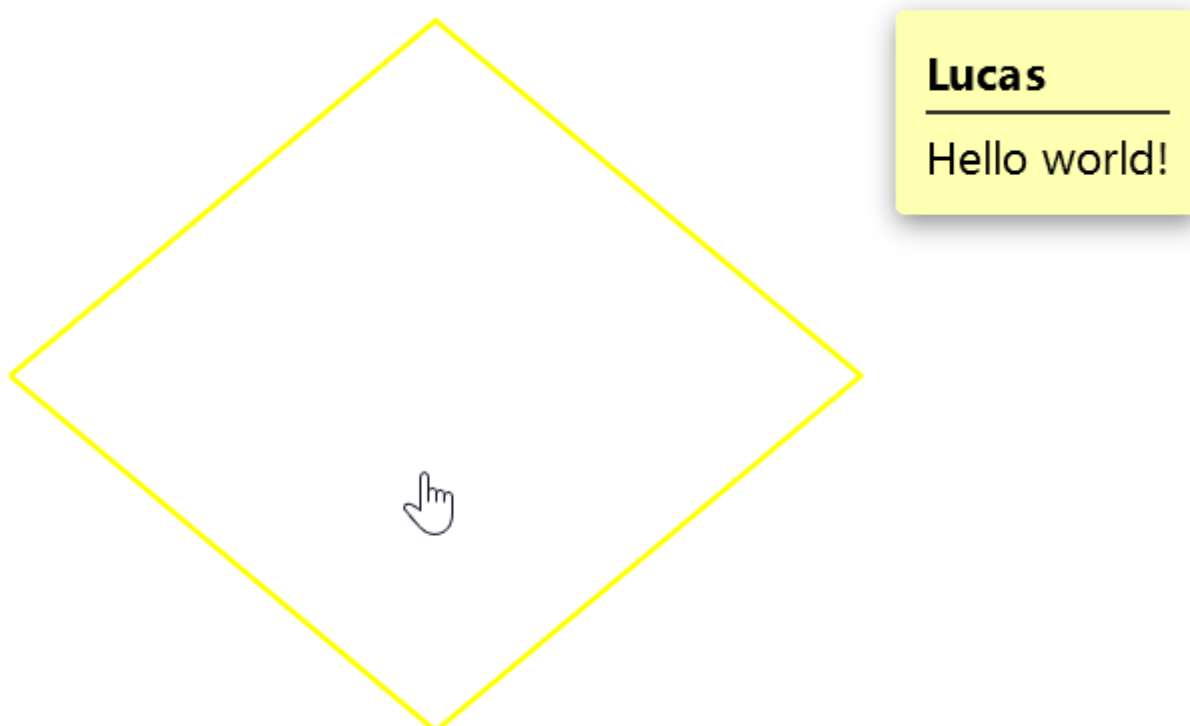
6.3.4 Ink annotations

Those annotations allow to draw paths around parts of a document to highlight them:

```
from fpdf import FPDF

pdf = FPDF()
pdf.ink_annotation([(100, 200), (200, 100), (300, 200), (200, 300), (100, 200)],
                  title="Lucas", contents="Hello world!")
pdf.output("ink_annotation_demo.pdf")
```

Rendering by Firefox internal PDF viewer:



Method documentation: [FPDF.ink_annotation](#)

6.3.5 File attachments

cf. the dedicated page: [File attachments](#)

6.3.6 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, text="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, text=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, text="Last page")
pdf.output("named_actions.pdf")
```

6.3.7 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, text=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")
```

6.4 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.

6.4.1 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(text="Page 1")
pdf.page_duration = .5
pdf.add_page()
pdf.cell(text="Page 2")
pdf.add_page()
pdf.cell(text="Page 3")
pdf.output("presentation.pdf")
```

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

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

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

6.5 Document outline & table of contents

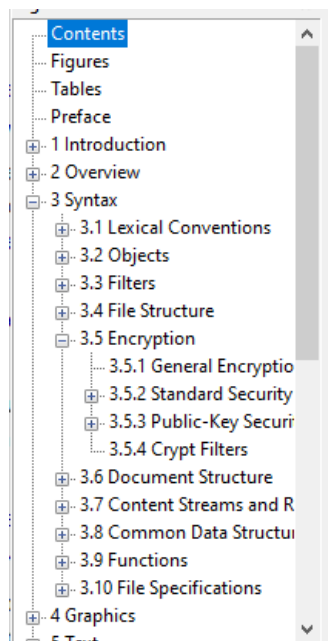
Quoting [Wikipedia](#), a **table of contents** is:

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](#):



PDF Reference

sixth edition

Adobe® Portable Document Format

Version 1.7

November 2006

Adobe Systems Incorporated

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.

6.5.1 With HTML

When using `FPDF.write_html`, 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`:

```
from fpdf import FPDF, HTML2FPDF

class CustomHTML2FPDF(HTML2FPDF):
    def render_toc(self, pdf, outline):
        pdf.cell(text='Table of contents:', new_x="LMARGIN", new_y="NEXT")
```



```

        for section in outline:
            pdf.cell(text=f'* {section.name} (page {section.page_number})', new_x="LMARGIN", new_y="NEXT")

class PDF(FPDF):
    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")

```

6.5.2 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 [simple_outline.pdf](#).

Similarly, `test_html_toc` generates [test_html_toc.pdf](#).

6.6 Encryption

New in  2.6.1

A PDF document can be encrypted to protect access to its contents.

An owner password is mandatory. Using the owner password anyone can perform any change on the document, including removing all encryption and access permissions.

The optional parameters are user password, access permissions and encryption method.

6.6.1 Password locking

User password is optional. If none is provided the document content is accessible for everyone.

If a user password is set, the content of the document will be encrypted and a password prompt displayed when a user opens the document. The document will only be displayed after either the user or owner password is entered.

```
pdf.set_encryption(
    owner_password="foo",
    user_password="bar"
)
```

6.6.2 Access permissions

Using access permissions flags you can restrict how the user interact with the document. The available access permission flags are:

- `PRINT_LOW_RES` Print the document, limiting the quality of the printed version.
- `PRINT_HIGH_RES` Print the document at the highest quality.
- `MODIFY` Modify the contents of the document.
- `COPY` Copy or extract text and graphics from the document.
- `ANNOTATION` Add or modify text annotations.
- `FILL_FORMS` Fill in existing interactive form fields.
- `COPY_FOR_ACCESSIBILITY` Extract text and graphics in support of accessibility to users with disabilities
- `ASSEMBLE` Insert, rotate or delete pages and create bookmarks or thumbnail images.

The flags can be combined using `|` :

```
from fpdf import FPDF
from fpdf.enums import AccessPermission

pdf = FPDF()
pdf.add_page()
pdf.set_font("helvetica", size=12)
pdf.cell(text="hello world")

pdf.set_encryption(
    owner_password="98765421",
    permissions=AccessPermission.PRINT_LOW_RES | AccessPermission.PRINT_HIGH_RES
)

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

The method `all()` grants all permissions and `none()` denies all permissions.

```
pdf.set_encryption(
    owner_password="xyz",
    permissions=AccessPermission.all()
)
```

If no permission is specified it will default to `all()`.

6.6.3 Encryption method

There are 4 available encryption methods:

- `NO_ENCRYPTION` Data is not encrypted, only add the access permission flags.
- `RC4` (default) Default PDF encryption algorithm.
- `AES_128` Encrypts the data with 128 bit key AES algorithm. Requires the `cryptography` package.
- `AES_256` Encrypts the data with 256 bit key AES algorithm. Requires the `cryptography` package.

```
from fpdf import FPDF
from fpdf.enums import AccessPermission, EncryptionMethod

pdf = FPDF()
pdf.add_page()
pdf.set_font("helvetica", size=12)
pdf.cell(text="hello world")

pdf.set_encryption(
    owner_password="123",
    encryption_method=EncryptionMethod.AES_128,
    permissions=AccessPermission.none()
)

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

6.7 Signing

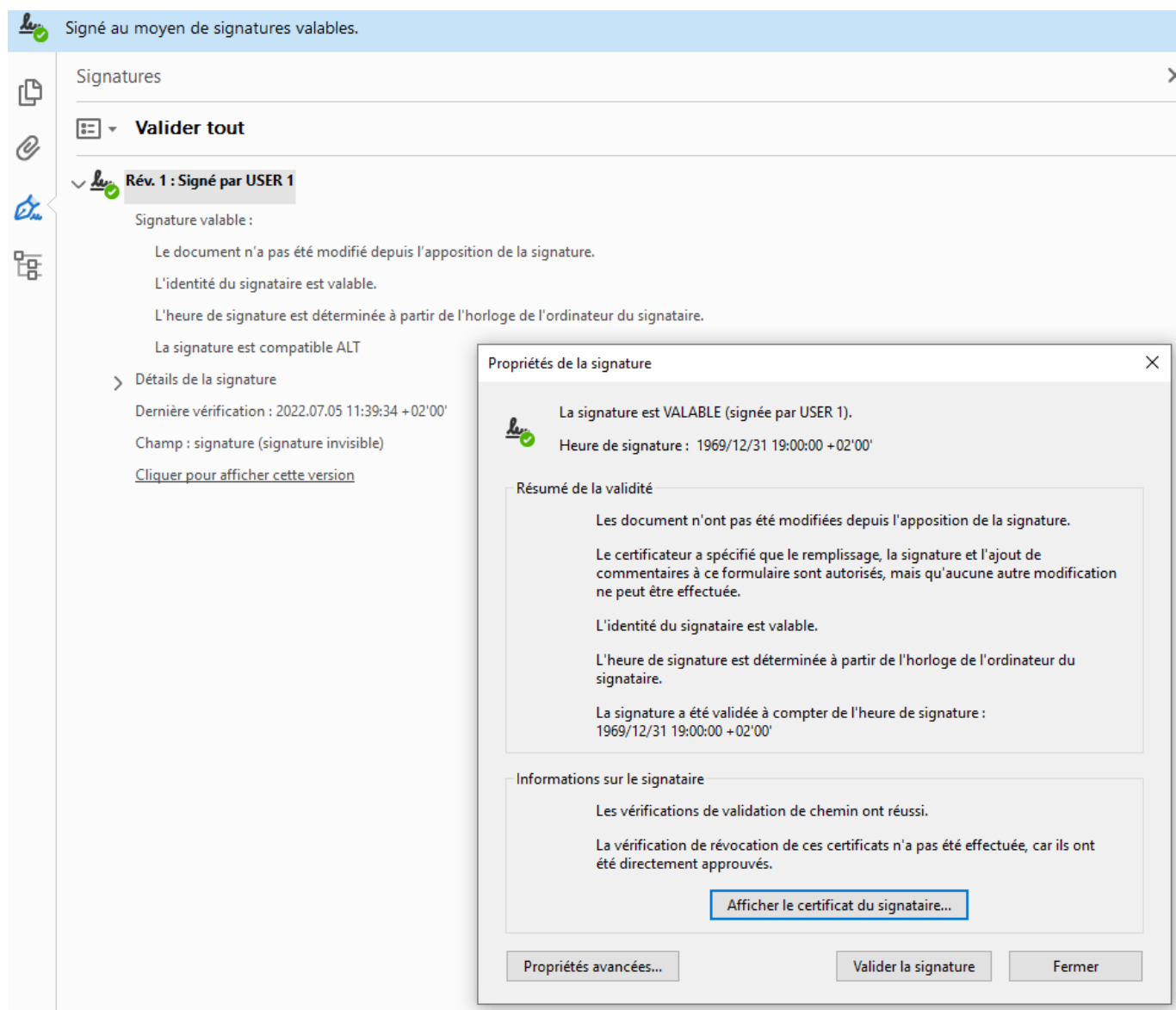
New in  2.5.6

A digital signature may be used to authenticate the identity of a user and the document's contents. It stores information about the signer and the state of the document when it was signed.

`fpdf2` allows to **sign** documents using **PKCS#12** certificates (RFC 7292).

The `endesive` package is **required** to do so.

```
pdf = FPDF()
pdf.add_page()
pdf.sign_pkcs12("certs.p12", password=b"1234")
pdf.output("signed_doc.pdf")
```



The lower-level `sign()` method allows to add a signature based on arbitrary key & certificates, not necessarily from a PKCS#12 file.

`endesive` also provides basic code to check PDFs signatures. [examples/pdf-verify.py](#) or the `check_signature()` function used in `fpdf2` unit tests can be good starting points for you, if you want to perform PDF signature control.

6.8 File attachments

6.8.1 Embedded file streams

Embedded file streams [allow] the contents of referenced files to be embedded directly within the body of the PDF file. This makes the PDF file a self-contained unit that can be stored or transmitted as a single entity.

`fpdf2` gives access to this feature through the method `embed_file()`:

```
pdf = FPDF()
pdf.add_page()
pdf.embed_file(__file__, desc="Source Python code", compress=True)
pdf.output("embedded_file.pdf")
```

6.8.2 Annotations

A file attachment annotation contains a reference to a file, which typically shall be embedded in the PDF file.

`fpdf2` gives access to this feature through the method `file_attachment_annotation()`:

```
pdf = FPDF()
pdf.add_page()
pdf.file_attachment_annotation(__file__, x=50, y=50)
pdf.output("file_attachment_annotation.pdf")
```

Resulting PDF: [file_attachment_annotation.pdf](#)

Browser PDF viewers do not usually display embedded files & file attachment annotations, so you may want to download this file and open it with your desktop PDF viewer in order to visualize the file attachments.



7. Mixing other libs

7.1 borb



Joris Schellekens made another excellent pure-Python library dedicated to reading & write PDF: [borb](#). He even wrote a very detailed e-book about it, available publicly there: [borb-examples](#).

The maintainer of `fpdf2` wrote an article comparing it with `borb`: [borb vs fpdf2](#).

7.1.1 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(text="Hello world!")

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

7.2 Combine with livereload

A nice feature of PDF readers is when they detect changes to the `.pdf` files open and automatically reload them in the viewer. Adobe Acrobat Reader **does not** provide this feature but other viewers offer it, like the free & open source [Sumatra PDF Reader](#) under Windows.

When using such PDF reader, it can be very useful to use a "watch" mode, so that every change to the Python code will trigger the regeneration of the PDF file.

The following script is an example of using [livereload](#) with `fpdf2` to do that. Launched without parameters, this script only generates a PDF document. But when launched with `--watch` as argument, it will detect changes to the Python script itself, and then reload itself with `xreload`, and finally regenerate the PDF document.

```
#!/usr/bin/env python3
# Script Dependencies:
#   fpdf2
#   livereload
#   xreload
import asyncio, logging, sys
from traceback import print_exc

from fpdf import FPDF
from livereload.watcher import get_watcher_class
from xreload import xreload

OUT_FILEPATH = "fpdf2-demo.pdf"

def build_pdf():
    pdf = FPDF()
    pdf.set_font("Helvetica", size=16)
    pdf.add_page()
    pdf.y += 50
    pdf.multi_cell(
        h=10,
        w=0,
        align="C",
        text="Hello fpdf2 user!
Launch this script with --watch
and then try to modify this text while the script is running"
    )
    pdf.output(OUT_FILEPATH)
    print(f"{OUT_FILEPATH} has been rebuilt")

async def start_watch_and_rebuild():
    logging.basicConfig(
        format="%(asctime)s %(name)s [%(levelname)s] %(message)s",
        datefmt="%H:%M:%S",
        level=logging.INFO,
    )
    logging.getLogger("livereload").setLevel(logging.INFO)
    watcher = get_watcher_class()()
    watcher.watch(__file__, build_pdf)
    print("Watcher started...")
    await watch_periodically(watcher)

async def watch_periodically(watcher, delay_secs=0.8):
    try:
        watcher.examine()
    except Exception:
        print_exc()
    await asyncio.sleep(delay_secs)
    xreload(sys.modules[__name__], new_annotations={"XRELOADED": True})
    await asyncio.create_task(watch_periodically(watcher))

# This conditional ensure that the code below
# does not get executed when calling xreload on this module:
if not __annotations__.get("XRELOADED"):
    build_pdf()
    # The --watch mode is very handy when using a PDF reader
    # that performs hot-reloading, like Sumatra PDF Reader:
    if "--watch" in sys.argv:
        asyncio.run(start_watch_and_rebuild())
```

Note that the module reloading mechanism provided by `xreload` has several limitations, cf. [xreload.py](#).

7.3 Combine with mistletoe to use Markdown

Several `fpdf2` methods allow Markdown syntax elements:

- `FPDF.cell()` has an optional `markdown=True` parameter that makes it possible to use `**bold**`, `_italics_` or `--underlined--` Markdown markers
- `FPDF.multi_cell()` & `FPDF.table()` methods have a similar feature

But `fpdf2` also allows for basic conversion **from HTML to PDF** (cf. [HTML](#)). This can be combined with the [mistletoe](#) library, that follows the [CommonMark specification](#), in order to generate **PDF documents from Markdown**:

```
from mistletoe import markdown

html = markdown(
    """
# Top title (ATX)

Subtitle (setext)
-----

### An even lower heading (ATX)

**Text in bold**

_Text in italics_

[This is a link](https://github.com/PyFPDF/fpdf2)

<https://py-pdf.github.io/fpdf2/>

This is an unordered list:
* an item
* another item

This is an ordered list:
1. first item
2. second item
3. third item with an unordered sublist:
    * an item
    * another item

Inline `code span`

A table:

| Foo | Bar | Baz |
| ---|:---|:--- |
| Foo | Bar | Baz |

Actual HTML:

<dl>
  <dt>Term1</dt><dd>Definition1</dd>
  <dt>Term2</dt><dd>Definition2</dd>
</dl>

Some horizontal thematic breaks:

***
---
___

![Alternate description](https://py-pdf.github.io/fpdf2/fpdf2-logo.png)
"""
)

from fpdf import FPDF

pdf = FPDF()
pdf.add_page()
pdf.write_html(html)
pdf.output("pdf-from-markdown.pdf")
```

7.3.1 Rendering unicode characters

```
from mistletoe import markdown

html = markdown(
    """
# Unicode:

| Emoji | Description |
```



```

| --- | - |
| 😄 | GRINNING FACE |
| 😊 | GRINNING FACE WITH SMILING EYES |
| 🤠 | SMILING FACE WITH HORNS |

# A checklist:

* ☐ item 1
* ☒ item 2
* ☐ item 3
"""
)

from fpdf import FPDF

pdf = FPDF()
pdf.add_font("DejaVuSans", fname="test/fonts/DejaVuSans.ttf")
pdf.add_font("DejaVuSans", fname="test/fonts/DejaVuSans-Bold.ttf", style="B")
pdf.set_font("DejaVuSans", size=24)
pdf.add_page()
pdf.write_html(html)
pdf.output("pdf-from-markdown.pdf")

```

Result:

Unicode:

Emoji



Description

GRINNING FACE

GRINNING FACE WITH
SMILING EYES

SMILING FACE WITH
HORNS

A checklist:

- ☐ item 1
- ☒ item 2
- ☐ item 3

7.4 Combine with pypdf

`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 `pypdf`, an actively-maintained library formerly known as `PyPDF2`.

7.4.1 Adding content onto an existing PDF page

In this code snippet, new content will be added on top of existing content:

```
#!/usr/bin/env python3
import io, sys

from fpdf import FPDF
from pypdf import PdfReader, PdfWriter

IN_FILEPATH = sys.argv[1]
OUT_FILEPATH = sys.argv[2]
ON_PAGE_INDEX = 0 # Index of the target page (starts at zero)

def new_content():
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("times", "B", 30)
    pdf.text(50, 150, "Hello World!")
    return io.BytesIO(pdf.output())

reader = PdfReader(IN_FILEPATH)
page_overlay = PdfReader(new_content()).pages[0]
reader.pages[ON_PAGE_INDEX].merge_page(page2=page_overlay)

writer = PdfWriter()
writer.append_pages_from_reader(reader)
writer.write(OUT_FILEPATH)
```

7.4.2 Adding a page to an existing PDF

```
#!/usr/bin/env python3
import io, sys

from fpdf import FPDF
from pypdf import PdfReader, PdfWriter

IN_FILEPATH = sys.argv[1]
OUT_FILEPATH = sys.argv[2]
ON_PAGE_INDEX = 2 # Index at which the page will be inserted (starts at zero)

def build_page():
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("times", "B", 19)
    pdf.text(50, 10, "Hello World!")
    return io.BytesIO(pdf.output())

writer = PdfWriter(clone_from=IN_FILEPATH)
new_page = PdfReader(build_page()).pages[0]
writer.insert_page(new_page, index=ON_PAGE_INDEX)
writer.write(OUT_FILEPATH)
```

7.4.3 Altering with pypdf a document generated with fpdf2

A document created with `fpdf2` can be edited with `pypdf` by passing its `.output()` to a `pypdf.PdfReader`:

```
import io
from fpdf import FPDF
from pypdf import PdfReader

pdf = FPDF()
pdf.add_page()
pdf.set_font('times', 'B', 19)
pdf.text(50, 10, 'Hello World!')
```

```
reader = PdfReader(io.BytesIO(pdf.output()))
```

7.5 Combine with pdfwr

`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 using `fpdf2` with `pdfwr`, 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.

Sadly, this library is not maintained anymore, cf. [pmaupin/pdfwr issue #232](#) & [sarnold/pdfwr issue #15](#).

7.5.1 Adding content onto an existing PDF page

```
#!/usr/bin/env python3
import sys
from fpdf import FPDF
from pdfwr import PageMerge, PdfReader, PdfWriter
from pdfwr.pagemerge import RectXObj

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

reader = PdfReader(IN_FILEPATH)
area = RectXObj(reader.pages[0])

def new_content():
    fpdf = FPDF(format=(area.w, area.h), unit="pt")
    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()
writer.pagearray = reader.Root.Pages.Kids
if writer.pagearray[0].Kids:
    writer.pagearray = writer.pagearray[0].Kids
PageMerge(writer.pagearray[ON_PAGE_INDEX]).add(
    new_content(), prepend=UNDERNEATH
).render()
writer.write(OUT_FILEPATH)
```

7.5.2 Adding a page to an existing PDF

```
#!/usr/bin/env python3
import sys

from fpdf import FPDF
from pdfwr 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 [pdfwr Pull Request #216](#). Until it is merged, you can install a forked version of `pdfwr` including the required patch:

```
pip install git+https://github.com/PyFPDF/pdfwr.git@addpage_at_index
```

7.5.3 Altering with pdfrw a document generated with fpdf2

A document created with `fpdf2` can be edited with `pdfrw` by passing its `.output()` to a `pdfrw.PdfReader`:

```
import io
from fpdf import FPDF
from pdfrw import PdfReader

pdf = FPDF()
pdf.add_page()
pdf.set_font('times', 'B', 19)
pdf.text(50, 10, 'Hello World!')

reader = PdfReader(io.BytesIO(pdf.output()))
```

7.6 Matplotlib, Pandas, Plotly, Pygal

7.7 Templating with Jinja

[Jinja](#) is a fast, expressive, extensible templating engine.

7.7.1 Combining Jinja & write_html

```
from fpdf import FPDF
from jinja2 import Environment

template = Environment().from_string("""
<h1>{{ title | escape }}</h1>
<ul>
{% for item in items %}
  <li>{{ item }}</li>
{% endfor %}
</ul>
""")

title = "HTML & Jinja demo"
items = [
    "FIRST",
    "SECOND",
    "LAST"
]

pdf = FPDF()
pdf.add_page()
pdf.write_html(template.render(**globals()))
pdf.output("templating_with_jinja.pdf")
```

More details about the supported HTML features: [HTML](#)

7.8 Usage in web APIs

Note that `FPDF` instance objects are not designed to be reusable: **content cannot be added** once `output()` has been called.

Hence, even if the `FPDF` class should be thread-safe, we recommend that you either **create an instance for every request**, or if you want to use a global / shared object, to only store the bytes returned from `output()`.

7.8.1 Django

Django is:

a high-level Python web framework that encourages rapid development and clean, pragmatic design

There is how you can return a PDF document from a [Django view](#):

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

def report(request):
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("Helvetica", size=24)
    pdf.cell(text="hello world")
    return HttpResponse(bytes(pdf.output()), content_type="application/pdf")
```

7.8.2 Flask

Flask is a micro web framework written in Python.

The following code can be placed in a `app.py` file and launched using `flask run`:

```
from flask import Flask, make_response
from fpdf import FPDF

app = Flask(__name__)

@app.route("/")
def hello_world():
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("Helvetica", size=24)
    pdf.cell(text="hello world")
    response = make_response(pdf.output())
    response.headers["Content-Type"] = "application/pdf"
    return response
```

7.8.3 AWS lambda

The following code demonstrates some minimal [AWS lambda handler function](#) that returns a PDF file as binary output:

```
from base64 import b64encode
from fpdf import FPDF

def handler(event, context):
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("Helvetica", size=24)
    pdf.cell(text="hello world")
    return {
        'statusCode': 200,
        'headers': {
            'Content-Type': 'application/json',
        },
        'body': b64encode(pdf.output()).decode('utf-8'),
        'isBase64Encoded': True
    }
```


This AWS lambda function can then be linked to a HTTP endpoint using [API Gateway](#), or simply exposed as a [Lambda Function URL](#). More information on those pages:

- [Tutorial: Creating a Lambda function with a function URL](#)
- [Return binary media from a Lambda](#)

For reference, the test lambda function was initiated using the following [AWS CLI](#) commands:

Creating & uploading a lambda layer

```
pyv=3.8
pip${pyv} install fpdf2 -t python/lib/python${pyv}/site-packages/
# We use a distinct layer for Pillow:
rm -r python/lib/python${pyv}/site-packages/{PIL,Pillow}*
zip -r fpdf2-deps.zip python > /dev/null
aws lambda publish-layer-version --layer-name fpdf2-deps \
  --description "Dependencies for fpdf2 lambda" \
  --zip-file fileb://fpdf2-deps.zip --compatible-runtimes python${pyv}
```

Creating the lambda

```
AWS_ACCOUNT_ID=...
AWS_REGION=eu-west-3
zip -r fpdf2-test.zip lambda.py
aws lambda create-function --function-name fpdf2-test --runtime python${pyv} \
  --zip-file fileb://fpdf2-test.zip --handler lambda.handler \
  --role arn:aws:iam::${AWS_ACCOUNT_ID}:role/lambda-fpdf2-role \
  --layers arn:aws:lambda:${AWS_REGION}:770693421928:layer:Klayers-python${pyv}/.-Pillow:15 \
  arn:aws:lambda:${AWS_REGION}:${AWS_ACCOUNT_ID}:layer:fpdf2-deps:1
aws lambda create-function-url-config --function-name fpdf2-test --auth-type NONE
```

Those commands do not cover the creation of the `lambda-fpdf2-role` role, nor configuring the lambda access permissions, for example with a `FunctionURLAllowPublicAccess` resource-based policy.

7.8.4 streamlit

[streamlit](#) is:

a Python library that makes it easy to create and share custom web apps for data science

The following code demonstrates how to display a PDF and add a button allowing to download it:

```
from base64 import b64encode
from fpdf import FPDF
import streamlit as st

st.title("Demo of fpdf2 usage with streamlit")

@st.cache
def gen_pdf():
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("Helvetica", size=24)
    pdf.cell(text="hello world")
    return bytes(pdf.output())

# Embed PDF to display it:
base64_pdf = b64encode(gen_pdf()).decode("utf-8")
pdf_display = f'<embed src="data:application/pdf;base64,{base64_pdf}" width="700" height="400" type="application/pdf">'
st.markdown(pdf_display, unsafe_allow_html=True)

# Add a download button:
st.download_button(
    label="Download PDF",
    data=gen_pdf(),
    file_name="file_name.pdf",
    mime="application/pdf",
)
```

7.8.5 FastAPI

FastAPI is:

a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints.

The following code shows how to generate a PDF file via a POST endpoint that receives a JSON object. The JSON object can be used to write into the PDF file. The generated PDF file will be returned back to the user/frontend as the response.

```
from fastapi import FastAPI, Request, Response, HTTPException, status
from fpdf import FPDF

app = FastAPI()

@app.post("/send_data", status_code=status.HTTP_200_OK)
async def create_pdf(request: Request):
    """
    POST endpoint that accepts a JSON object
    This endpoint returns a PDF file as the response
    """
    try:
        # data will read the JSON object and can be accessed like a Python Dictionary
        # The contents of the JSON object can be used to write into the PDF file (if needed)
        data = await request.json()

        # Create a sample PDF file
        pdf = FPDF()
        pdf.add_page()
        pdf.set_font("Helvetica", size=24)
        pdf.cell(text="hello world")
        # pdf.cell(text=data["content"]) # Using the contents of the JSON object to write into the PDF file
        # Use str(data["content"]) if the content is non-string type

        # Prepare the filename and headers
        filename = "<file_name_here>.pdf"
        headers = {
            "Content-Disposition": f"attachment; filename={filename}"
        }

        # Return the file as a response
        return Response(content=bytes(pdf.output()), media_type="application/pdf", headers=headers)

    except Exception as e:
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail=str(e))
```

7.8.6 Jupyter

Check [tutorial/notebook.ipynb](#)

7.8.7 web2py

Usage of the original PyFPDF lib with [web2py](#) is described here: <https://github.com/reingart/pyfpdf/blob/master/docs/Web2Py.md>

[v1.7.2](#) of PyFPDF is included in [web2py](#) since release [1.85.2](#): <https://github.com/web2py/web2py/tree/master/gluon/contrib/fpdf>

7.9 Database storage

7.9.1 SQLAlchemy

The following snippet demonstrates how to store PDFs built with `fpdf2` in a database, and then retrieve them, using [SQLAlchemy](#):

```
from fpdf import FPDF
from sqlalchemy import create_engine, Column, Integer, LargeBinary, String
from sqlalchemy.orm import declarative_base, sessionmaker

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    pdf = Column(LargeBinary)

engine = create_engine('sqlite:///memory:', echo=True)
Base.metadata.create_all(engine)

pdf = FPDF()
pdf.add_page()
pdf.set_font("Helvetica", size=24)
pdf.cell(text="My name is Bobby")
new_user = User(name="Bobby", pdf=pdf.output())

Session = sessionmaker(bind=engine)
session = Session()

session.add(new_user)

user = session.query(User).filter_by(name="Bobby").first()
with open("user.pdf", "wb") as pdf_file:
    pdf_file.write(user.pdf)
```

Note that storing large binary data in a database is usually not recommended... You might be better off dynamically generating your PDFs from structured data in your database.

8. Development

8.1 Development

This page has summary information about developing the fpdf2 library.

- [Development](#)
- [Repository structure](#)
- [Installing fpdf2 from a local git repository](#)
- [Code auto-formatting](#)
- [Linting](#)
- [Pre-commit hook](#)
- [Testing](#)
- [Running tests](#)
- [Why is a test failing?](#)
- [assert_pdf_equal & writing new tests](#)
- [Generating PDF files for testing](#)
- [Visually comparing all PDF reference files modified on a branch](#)
- [Testing performances](#)
- [Code speed & profiling](#)
- [Tracking memory usage](#)
- [Non-regression performance tests](#)
- [GitHub pipeline](#)
- [Release checklist](#)
- [Documentation](#)
- [PDF spec & new features](#)
- [Useful tools to manipulate PDFs](#)
- [qpdf](#)
- [set_pdf_xref.py](#)

8.1.1 Repository structure

- `.github/` - GitHub Actions configuration
- `docs/` - documentation folder
- `fpdf/` - library sources
- `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](#)
- `.banditrc.yml` - configuration for [bandit](#)
- `.pylintrc` - configuration for [Pylint](#)

8.1.2 Installing fpdf2 from a local git repository

```
pip install --editable $path/to/fpdf/repo
```

This will link the installed Python package to the repository location, basically meaning any changes to the code package will get reflected directly in your environment.

8.1.3 Code auto-formatting

We use [black](#) as a code prettifier. This "*uncompromising Python code formatter*" must be installed in your development environment in order to auto-format source code before any commit:

```
pip install black
black . # inside fpdf2 root directory
```

8.1.4 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.

8.1.5 Pre-commit hook

This project uses [git pre-commit hooks](#): <https://pre-commit.com>

Those hooks are configured in `.pre-commit-config.yaml`.

They are intended to abort your commit if `pylint` found issues or `black` detected 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.

To install pre-commit hooks on your computer, run:

```
pip install pre-commit
pre-commit install
```

8.1.6 Testing

Running tests

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

You may also need to install [SWIG](#) and [Ghostscript](#), because they are dependencies for `camelot`, a library for table extraction in PDF that we test in `test/table/test_table_extraction.py`. Those tests will always be executed by the GitHub Actions pipeline, so you can also not bother installing those tools and skip those tests by running `pytest -k "not camelot"`.

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).

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 visualize to spot differences:

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

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.

Generating PDF files for testing

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

```
import fpdf

svg = fpdf.svg.SVGObject.from_file("path/to/file.svg")
pdf = fpdf.FPDF(unit="pt", format=(svg.width, svg.height))
pdf.add_page()
svg.draw_to_page(pdf)

assert_pdf_equal(
    pdf,
    "path/for/pdf/output.pdf",
    "path/for/pdf/",
    generate=True
)
```

Visually comparing all PDF reference files modified on a branch

This script will build and serve a single HTML page containing all PDF references file modified on your current `git` branch, and render them side by side with the PDF file from the `master` branch, so that you can quickly scroll and check for visible differences:

```
scripts/compare-changed-pdfs.py
```

8.1.7 Testing performances

Code speed & profiling

First, try to write a really **MINIMAL** Python script that focus strictly on the performance point you are investigating. Try to choose the input dataset so that the script execution time is between 1 and 15 seconds.

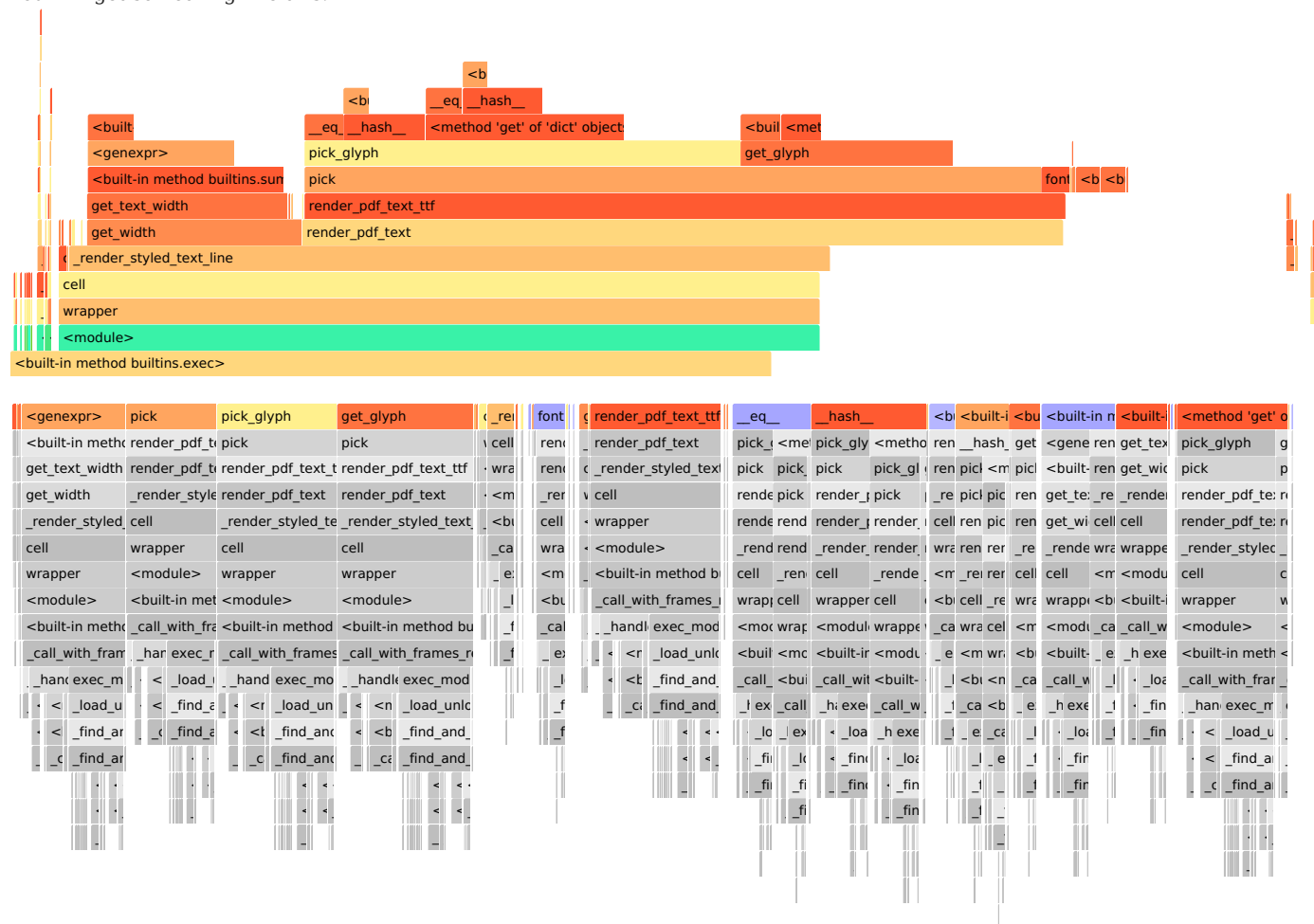
Then, you can use `cProfile` to profile your code and produce a `.pstats` file:

```
python -m cProfile -o profile.pstats script.py
```

Finally, you can quickly convert this `.pstats` file into a SVG flamegraph using `flameprof`:

```
pip install flameprof
flameprof profile.pstats > script-flamegraph.svg
```

You will get something like this:



Source GitHub thread where this was produced: [issue #907](#)

Tracking memory usage

A good way to track memory usage is to insert calls to `fpdf.util.print_mem_usage()` in the code you are investigating. This function will display the current process **resident set size (RSS)** which is currently, to the maintainer knowledge, one of the best way to get an accurate measure of Python scripts memory usage.

There is an example of using this function to track `fpdf2` memory usage in this issue comment: [issue #641](#). This thread also includes some tests of other libs & tools to track memory usage.

Non-regression performance tests

We try to have a small number of unit tests that ensure that the library performances do not degrade over time, when refactoring are made and new features added.

We have 2 test decorators to help with this:

- `@ensure_exec_time_below`
- `@ensure_rss_memory_below`

As of `fpdf2` v2.7.6, we only keep 3 non-regression performance tests:

- `test_intense_image_rendering()` in `test_perfs.py`
- `test_charmap_first_999_chars()` in `test_charmap.py`
- `test_cell_speed_with_long_text()` in `test_cell.py`

8.1.8 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.

Release checklist

1. complete `CHANGELOG.md` and add the version & date of the new release
2. bump `FPDF_VERSION` in `fpdf/fpdf.py`. Also (optionnal, once every year), update `contributors/contributors-map-small.png` based on <https://py-pdf.github.io/fpdf2/contributors.html>
3. update the `announce` block in `docs/overrides/main.html` to mention the new release
4. `git commit` & `git push` (if editing in a fork: submit and merge a PR)
5. check that [the GitHub Actions succeed](#), and that [a new release appears on Pypi](#)
6. perform a [GitHub release](#), taking the description from the `CHANGELOG.md`. It will create a new `git` tag.
7. Announce the release on [r/pythonnews](#), and add an announcement to the documentation website: [docs/overrides/main.html](https://py-pdf.github.io/fpdf2/docs/overrides/main.html)

8.1.9 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 :
```

8.1.10 PDF spec & new features

The **PDF 1.7 spec** is available on Adobe website: [PDF32000_2008.pdf](#).

The **PDF 2.0 spec** is available on the [Adobe website](#) or on the [PDF Association website](#)

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.

8.1.11 Useful tools to manipulate PDFs

qpdf

[qpdf](#) is a very powerful tool to analyze PDF documents.

One of its most useful features is the [QDF mode](#) that can convert any PDF file to a human-readable, decompressed & annotated new PDF document:

```
qpdf --qdf doc.pdf doc-qdf.pdf
```

This is extremely useful to peek into the PDF document structure.

set_pdf_xref.py

[set_pdf_xref.py](#) is a small Python script that can **rebuild a PDF xref table**.

This is very useful, as a PDF with an invalid xref cannot be opened. An xref table is basically an index of the document internal sections. When manually modifying a PDF file (for example one produced by `qpdf --qdf`), if the characters count in any of its sections changes, the xref table must be rebuilt.

With `set_pdf_xref.py doc.pdf --inplace`, you can change some values inside any PDF file, and then quickly make it valid again to be viewed in a PDF viewer.

8.2 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 %(name)s [%(levelname)s] %(message)s",
                    datefmt="%H:%M:%S", level=logging.DEBUG)
```

Example output using the [Tutorial](#) first code snippet:

```
19:25:24 fpdf.output [DEBUG] Final size summary of the biggest document sections:
19:25:24 fpdf.output [DEBUG] - pages: 223.0B
19:25:24 fpdf.output [DEBUG] - fonts: 102.0B
```

8.2.1 fonttools verbose logs

Since `fpdf2` v2.5.7, verbose **INFO** logs are generated by `fonttools`, a library we use to parse font files:

```
fontTools.subset [INFO] maxp pruned
fontTools.subset [INFO] cmap pruned
fontTools.subset [INFO] post pruned
fontTools.subset [INFO] EBDT dropped
fontTools.subset [INFO] EBLC dropped
fontTools.subset [INFO] GDEF dropped
fontTools.subset [INFO] GPOS dropped
fontTools.subset [INFO] GSUB dropped
fontTools.subset [INFO] DSIG dropped
fontTools.subset [INFO] name pruned
fontTools.subset [INFO] glyf pruned
fontTools.subset [INFO] Added gid0 to subset
fontTools.subset [INFO] Added first four glyphs to subset
fontTools.subset [INFO] Closing glyph list over 'glyf': 25 glyphs before
fontTools.subset [INFO] Glyph names: ['.notdef', 'b', 'braceleft', 'braceright', 'd', 'e', 'eight', 'five', 'four', 'glyph1', 'glyph2', 'h', 'l', 'n', 'nine', 'o', 'one', 'r', 'seven', 'six', 'space', 'three', 'two', 'w', 'zero']
fontTools.subset [INFO] Glyph IDs: [0, 1, 2, 3, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 69, 71, 72, 75, 79, 81, 82, 85, 90, 94, 96]
fontTools.subset [INFO] Closed glyph list over 'glyf': 25 glyphs after
fontTools.subset [INFO] Glyph names: ['.notdef', 'b', 'braceleft', 'braceright', 'd', 'e', 'eight', 'five', 'four', 'glyph1', 'glyph2', 'h', 'l', 'n', 'nine', 'o', 'one', 'r', 'seven', 'six', 'space', 'three', 'two', 'w', 'zero']
fontTools.subset [INFO] Glyph IDs: [0, 1, 2, 3, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 69, 71, 72, 75, 79, 81, 82, 85, 90, 94, 96]
fontTools.subset [INFO] Retaining 25 glyphs
fontTools.subset [INFO] head subsetting not needed
fontTools.subset [INFO] hhea subsetting not needed
fontTools.subset [INFO] maxp subsetting not needed
fontTools.subset [INFO] OS/2 subsetting not needed
fontTools.subset [INFO] hmtx subsetting
fontTools.subset [INFO] cmap subsetting
fontTools.subset [INFO] fpgm subsetting not needed
fontTools.subset [INFO] prep subsetting not needed
fontTools.subset [INFO] cvt subsetting not needed
fontTools.subset [INFO] loca subsetting not needed
fontTools.subset [INFO] post subsetting
fontTools.subset [INFO] name subsetting not needed
fontTools.subset [INFO] glyf subsetting
fontTools.subset [INFO] head pruned
fontTools.subset [INFO] OS/2 Unicode ranges pruned: [0]
fontTools.subset [INFO] glyf pruned
```

You can easily suppress those logs with this single line of code:

```
logging.getLogger('fontTools.subset').level = logging.WARN
```

Similarly, you can omit verbose logs from `fontTools.ttLib.ttFont`:

```
logging.getLogger('fontTools.ttLib.ttFont').level = logging.WARN
```

8.2.2 Warning logs for unsupported SVG features

The `fpdf.svg` module produces `WARNING` log messages for **unsupported** SVG tags & attributes. If need be, you can suppress those logs:

```
logging.getLogger("fpdf.svg").propagate = False
```

9. History

This project, `fpdf2` is a *fork* of the `PyFPDF` project, which can still be found [on GitHub at reingart/pyfpdf](#), but has been totally inactive since January 2018, and has not seen any new release since 2015.

About the original `PyFPDF` lib:

This project started as a Python fork of the `FPDF` PHP library, ported to Python by Max Pat in 2006: <http://www.fpdf.org/dl.php?id=94>. Later, code for native reading TTF fonts was added. The project 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. Until 2015 the code was developed at [Google Code](#): you can still access the [old issues](#), and [old wiki](#).

9.1 How fpdf2 came to be

During the spring of 2016, David Ankin ([@alexanderankin](#)) started a fork of `PyFPDF`, and added the first commit of what became `fpdf2`: [bd608e4](#). On May of 2017, the first release of `fpdf2` was published on Pypi: [v2.0.0](#).

On 2020, the first PRs were merged from external contributors. At the end of the year, Lucas Cimon ([@Lucas-C](#)) started contributing several improvements, in order to use `fpdf2` for his [Undying Dusk](#) project. [Version 2.1.0 was released](#) and on 2021/01/10 `fpdf2` was moved to a dedicated `PyFPDF` GitHub organization, and [@Lucas-C](#) became another maintainer of the project.

On 2023/08/04, `fpdf2` moved to the `py-pdf` organization: <https://github.com/py-pdf/fpdf2>. The context for this move can be found there: [discussion #752](#). On this date, the `PyFPDF` GitHub organization has been **archived**. The same month, Georg Mischler ([@gmischler](#)) and Anderson Herzogenrath da Costa ([@andersonhc](#)) joined the project as new maintainers.


9.2 Compatibility between PyFPDF & fpdf2

`fpdf2` aims to be fully compatible with `PyFPDF` code.

The notable exceptions are:

- for the `cell()` method, the default value of `h` has changed. It used to be `0` and is now set to the current value of `FPDF.font_size`
- the font caching mechanism, that used the `pickle` module, has been **removed**, for security reasons, and because it provided little performance gain, and only for specific use cases - cf. [issue #345](#).
- [Template](#) elements now have a **transparent background** by default, instead of white

Some features are also **deprecated**. As of version 2.7.5 they **still work** but **generate a warning** when used:

-  `FPDF.rotate()` can produce malformed PDFs: use `FPDF.rotation()` instead
- `FPDF.set_doc_option()`: simply set the `.core_fonts_encoding` property as a replacement
- `FPDF.dashed_line()`: use `FPDF.set_dash_pattern()` and the normal drawing operations instead
- the `font_cache_dir` parameter of `FPDF()` constructor, that is currently ignored
- the `uni` parameter of `FPDF.add_font()`, that is currently ignored: if the value of the `fname` argument passed to `add_font()` ends with `.ttf`, it is considered a TrueType font
- the `type` parameter of `FPDF.image()`, that is currently ignored
- the `dest` parameter of `FPDF.output()`, that is currently ignored
- the `ln` parameter of `FPDF.multi_cell()`: use `new_x=` & `new_y=` instead
- the `split_only` parameter of `FPDF.multi_cell()`: use `dry_run=True` and `output="LINES"` instead
- the `HTMLMixin` class: you can now directly use the `FPDF.write_html()` method
- the `infile` parameter of `Template()` constructor, that is currently ignored
- the parameters `x/y/w/h` of `code39` elements provided to the `Template` system: please use `x1/y1/y2/size` instead
- the `dest` parameter of `Template.render()`, that is currently ignored

Note that `DeprecationWarning` messages are not displayed by Python by default. To get warned about deprecated features used in your code, you must execute your scripts with the `-Wd` option (cf. [documentation](#)), or enable them programmatically with `warnings.simplefilter('default', DeprecationWarning)`.

10. FAQ

See [Project Home](#) for an overall introduction.

- [FAQ](#)
- [What is fpdf2?](#)
- [What is this library not?](#)
- [How does this library compare to ...?](#)
- [What does the code look like?](#)
- [Does this library have any framework integration?](#)
- [What is the development status of this library?](#)
- [What is the license of this library \(fpdf2\)?](#)

10.1 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>

10.2 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.

10.3 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
1,6M    fpdf

$ scc fpdf
```

Language	Files	Lines	Blanks	Comments	Code Complexity
Python	21	16879	480	571	15828 462

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](#).

10.4 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, text=title, border=0,
            new_x="LMARGIN", new_y="NEXT", align='C', fill=False)
    pdf.set_font('Times', 'B', 15)
    pdf.cell(w=0, h=6, text=heading, border=0,
            new_x="LMARGIN", new_y="NEXT", align='L', fill=False)
    pdf.set_font('Times', '', 12)
    pdf.multi_cell(w=0, h=5, text=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
```

10.5 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.

10.6 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 developed regularly.

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

10.7 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 original `fpdf.py` library was 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](#).