fpdf2 manual

A minimalist PDF creation library for Python

Table of contents

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` `v2.5.2`

```python
from fpdf import FPDF

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

1.1 Main features

- Easy to use, with a user-friendly API, and easy to extend
- Python 3.6+ support
- Unicode (UTF-8) TrueType font subset embedding (Central European, Cyrillic, Greek, Baltic, Thai, Chinese, Japanese, Korean, Hindi and almost any other language in the world)
- 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

- Cell / multi-cell / plaintext writing, automatic page breaks

- Basic conversion from HTML to PDF

- A templating system to render PDFs in batchs

- Images & links alternative descriptions, for accessibility

- Table of contents & document outline

- Annotations, including text highlights

- Presentation mode with control over page display duration & transitions

- Optional basic Markdown-like styling: `**bold**`, `__italics__`

- Only has 2 dependencies: Pillow & defusedxml

- Unit tests with `qpdf` -based PDF diffing

- Many example scripts available throughout this documentation, including usage examples with Django, Flask, streamlit... : Usage in web APIs

- PDF samples validation using 3 different checkers:

FPDF original features:

- Choice of measurement unit, page format and margins

- Page header and footer management

- Automatic page break, line break and text justification

- Image, colors and links support

- Page compression

1.2 Tutorials

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

1.3 Installation

From PyPI:

```
pip install fpdf2
```

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

```
pip install git+https://github.com/PyFPDF/fpdf2.git@master
```

Developement: 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

- 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

1.4.3 Related

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

- Create PDFs with Python : a series of tutorial videos by bvalgard

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

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

2.1 Tutorial

Methods full documentation: `fpdf.FPDF` API doc

2.1.1 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=XPos.LMARGIN, new_y=YPos.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.

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

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

```python
from fpdf import FPDF


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

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


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

Resulting PDF

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

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

To print the page number, a null value is passed as the cell width. It means that the cell should extend up to the right margin of the page; it is handy to center text. The current page number is returned by the page_no method; as for the total number of pages, it is obtained by means of the special value `{nb}` which will be substituted on document closure (this special value can 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.

2.1.3 Tuto 3 - Line breaks and colors

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

```python
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)
```

```python
        # Setting colors for frame, background and text:
        self.set_draw_color(0, 80, 180)
        self.set_fill_color(230, 230, 0)
        self.set_text_color(220, 50, 50)
        # Setting thickness of the frame (1 mm)
        self.set_line_width(1)
        # Printing title:
        self.cell(width, 9, self.title, 1, 1, "C", True)
        # Performing a line break:
        self.ln(10)

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

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

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

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


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

Resulting PDF

Jules Verne text

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

The method used to print the paragraphs is multi_cell. Text is justified by default. Each time a line reaches the right extremity of the cell or a carriage return character (\n) is met, a line break is issued and a new cell automatically created under the current one. An automatic break is performed at the location of the nearest space or soft-hyphen (\u00ad) character before the right limit. A soft-hyphen will be replaced by a normal hyphen when triggering a line break, and ignored otherwise.

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

2.1.4 Tuto 4 - Multi Columns

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

```python
from fpdf import FPDF


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

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

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

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

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

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

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

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


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

Resulting PDF

Jules Verne text

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

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

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

2.1.5 Tuto 5 - Creating Tables

This tutorial will explain how to create tables easily.

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

```python
import csv
from fpdf import FPDF


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

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

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


def load_data_from_csv(csv_filepath):
    headings, rows = [], []
    with open(csv_filepath, encoding="utf8") as csv_file:
        for row in csv.reader(csv_file, delimiter=","):
            if not headings:  # extracting column names from first row:
                headings = row
            else:
                rows.append(row)
    return headings, rows


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

Resulting PDF - Countries text

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

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

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

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

2.1.6 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.

```python
from fpdf import FPDF, HTMLMixin


class MyFPDF(FPDF, HTMLMixin):
    pass


pdf = MyFPDF()

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

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

Resulting PDF - fpdf2-logo

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

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

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

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

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

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

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

2.2 Kurzanleitung

Vollständige Dokumentation der Methoden: `fpdf.FPDF` API doc

- Kurzanleitung
- Lektion 1 - Minimalbeispiel
- Lektion 2 - Kopfzeile, Fußzeile, Seitenumbruch und Bild
- Lektion 3 - Zeilenumbrüche und Farben
- Lektion 4 - Mehrspaltiger Text
- Lektion 5 - Tabellen erstellen
- Lektion 6 - Links erstellen und Textstile mischen

2.2.1 Lektion 1 - Minimalbeispiel

Beginnen wir mit dem Klassiker:

```python
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")
```

Erzeugtes PDF

Nachdem wir die Bibliothek eingebunden haben, erstellen zuerst wir ein `FPDF` Objekt. Der `FPDF` Konstruktor wird hier mit den Standardwerten verwendet: Das Seitenformat wird auf A4-Hochformat gesetzt und als Maßeinheit Millimeter festgelegt.

Diese Werte hätten wir auch explizit angegeben können:

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

Es ist auch möglich, eine PDF-Datei im Querformat zu erstellen (`L`), sowie andere Seitenformate (`Letter` und `Legal`) und Maßeinheiten (`pt` , `cm` , `in`) zu verwenden.

Bisher haben wir dem Dokument noch keine Seite hinzugefügt. Um eine Seite hinzuzufügen, verwenden wir `add_page` . Der Ursprung der Koordinaten liegt in der oberen linken Ecke und die aktuelle Schreibposition ist standardmäßig jeweils 1 cm von den Rändern entfernt. Diese Randabstände können mit `set_margins` angepasst werden.

Bevor wir Text hinzufügen können, müssen wir zuerst mit `set_font` eine Schriftart festlegen, um ein gültiges Dokument zu erzeugen. Wir wählen Helvetica, fett in Schriftgröße 16 pt:

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

Anstelle von `B` hätten wir mit `I` kursiv , `U` unterstrichen oder durch die Übergabe einer leeren Zeichenkette einen "normale" Textstil wählen können. Beliebige Kombinationen der drei Werte sind zulässig. Beachte, dass die Schriftgröße in Punkt und nicht in Millimetern (oder einer anderen durch den Benutzer bei der Erstellung mit `unit=` festgelegten Maßeinheit) angegeben wird. Dies ist die einzige Ausnahme vom Grundsatz, dass immer die durch den Benutzer gewählte Maßeinheit bei der Festlegung von Positions- oder Größenangaben genutzt wird. Neben `Helvetica` stehen `Times` , `Courier` , `Symbol` und `ZapfDingbats` als Standardschriftarten zur Verfügung.

Wir können jetzt eine erste Textzelle mit `cell` einfügen. Eine Zelle ist ein rechteckiger Bereich - optional umrahmt - der Text enthalten kann. Sie wird an der jeweils aktuellen Schreibposition gerendert. Wir können die Abmessungen der Zelle, den Text und dessen Formatierung (zentriert oder ausgerichtet), einen ggf. gewünschten Rahmen und die Festlegung der neuen Schreibposition nach dem Schreiben der Zelle (rechts, unten oder am Anfang der nächsten Zeile) bestimmen.

Um einen Rahmen hinzuzufügen, würden wir die Methode folgendermaßen einbinden:

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

Um eine neue Zelle mit zentriertem Text hinzuzufügen und anschließend in die nächste Zeile zu springen, können wir Folgendes schreiben:

```
pdf.cell(60, 10, 'Powered by FPDF.', new_x=XPos.LMARGIN, new_y=YPos.NEXT, align='C')
```

Anmerkung: Der Zeilenumbruch kann auch mit `ln` erfolgen. Diese Methode erlaubt es, zusätzlich die Höhe des Umbruchs anzugeben.

Schließlich wird das Dokument mit `output` geschlossen und unter dem angegebenen Dateipfad gespeichert. Ohne Angabe eines Parameters liefert `output()` den PDF `bytearray`-Puffer zurück.

2.2.2 Lektion 2 - Kopfzeile, Fußzeile, Seitenumbruch und Bild

Hier ein zweiseitiges Beispiel mit Kopfzeile, Fußzeile und Logo:

```python
from fpdf import FPDF


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

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


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

Erzeugtes PDF

Dieses Beispiel verwendet die Methoden `header` und `footer`, um Kopf- und Fußzeilen zu verarbeiten. Sie werden jeweils automatisch aufgerufen. Die Methode 'header' direkt nach dem Hinzugügen einer neuen Seite, die Methode 'footer' wenn die Bearbeitung einer Seite durch das Hinzufügen einer weiteren Seite oder das Abspeichern des Dokuments abgeschlossen wird. Die Methoden existieren bereits in der Klasse FPDF, sind aber leer. Um sie zu nutzen, müssen wir die Klasse erweitern und sie überschreiben.

Das Logo wird mit der Methode `image` eingebunden, und auf der Seite durch die Angabe der Position der linken oberen Ecke und die gewünschte Bildbreite platziert. Die Höhe wird automatisch berechnet, um die Proportionen des Bildes zu erhalten.

Um die Seitenzahl einzufügenn, übergeben wir zuerst der Zelle einen Nullwert als Breite der Zelle. Das bedeutet, dass die Zelle bis zum rechten Rand der Seite reichen soll. Das ist besonders praktisch, um Text zu zentrieren. Die aktuelle Seitenzahl wird durch die Methode `page_no` ermittelt und in die Zelle geschrieben. Die Gesamtseitenzahl wird mit Hilfe des speziellen Platzhalterwertes `{nb}` ermittelt, der beim Schließen des Dokuments ersetzt wird aufgerufen. Beachte die Verwendung der Methode `set_y`, mit der du die vertikale Schreibposition an einer absoluten Stelle der Seite - von oben oder von unten aus - setzen kannst.

Eine weitere interessante Funktion wird hier ebenfalls verwendet: der automatische Seitenumbruch. Sobald eine Zelle eine festgelegte Grenze in der Seite überschreitet (standardmäßig 2 Zentimeter vom unteren Rand), wird ein Seitenumbruch durchgeführt und die Einstellungen der gewählten Schrift auf der nächsten Seite automatisch beibehalten. Obwohl die Kopf- und

Fußzeilen ihre eigene Schriftart (`Helvetica`) wählen, wird im Textkörper `Times` verwendet. Dieser Mechanismus der automatischen Übernahme der Einstellungen nach Seitenumbruch gilt auch für Farben und Zeilenbreite. Der Grenzwert, der den Seitenumbruch auslöst, kann mit `set_auto_page_break` festgelegt werden .

2.2.3 Lektion 3 - Zeilenumbrüche und Farben

Fahren wir mit einem Beispiel fort, das Absätze im Blocksatz ausgibt. Es demonstriert auch die Verwendung von Farben.

```python
from fpdf import FPDF


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

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

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

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

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


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

Resulting PDF

Jules Verne text

Die Methode `get_string_width` ermöglicht die Bestimmung die Breite des übergebenen Textes in der aktuellen Schriftart. Das Beispiel nutzt sie zur Berechnung der Position und der Breite des Rahmens, der den Titel umgibt. Anschließend werden die Farben mit `set_draw_color` , `set_fill_color` und und `set_text_color` gesetzt und die Linienstärke mit `set_line_width` auf 1 mm

(Abweichend vom Standardwert von 0,2) festgelegt. Schließlich geben wir die Zelle aus (Der letzte Parameter True zeigt an, dass der Hintergrund gefüllt werden muss).

Zur Erstellung von Absätzen wir die Methode `multi_cell` genutzt. Jedes Mal, wenn eine Zeile den rechten Rand der Zelle erreicht oder ein Zeilenumbruchzeichen `\\n` im Text erkannt wird, wird ein Zeilenumbruch durchgeführt und automatisch eine neue Zelle unterhalb der aktuellen Zelle erstellt. Der Text wird standardmäßig im Blocksatz ausgerichtet.

Es werden zwei Dokumenteigenschaften definiert: Titel (`set_title`) und Autor (`set_author`). Dokumenteneigenschaften können auf zwei Arten eingesehen werden. Man kann das Dokument mit dem Acrobat Reader öffnen und im Menü **Datei** die Option **Dokumenteigenschaften** auswählen. Alternativ kann man auch mit der rechten Maustaste auf das Dokument klicken und die Option Dokumenteigenschaften wählen.

2.2.4 Lektion 4 - Mehrspaltiger Text

Dieses Beispiel ist eine Abwandlung des vorherigen Beispiels und zeigt, wie sich Text über mehrere Spalten verteilen lässt.

```python
from fpdf import FPDF


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

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

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

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

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

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

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

```
        # Start back at first column:
        self.set_col(0)

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


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

Erzeugtes PDF

Jules Verne Text

Der Hauptunterschied zur vorherigen Lektion ist die Verwendung der Methoden `accept_page_break` und `set_col`.

Wird `accept_page_break` verwendet, wird die aktuelle Spaltennummer überprüft, sobald die Zelle den zur Auslösung eines Seitenumbruchs festgelegten Abstand zum unteren Seitenrand (Standard 2cm) überschreitet. Ist die Spaltennummer kleiner als 2 (wir haben uns entschieden, die Seite in drei Spalten zu unterteilen), wird die Methode `set_col` aufgerufen. Sie erhöht die Spaltennummer auf die nächsthöhere und setzt die Schreibposition auf den Anfang der nächsten Spalte, damit der Text dort fortgesetzt werden kann.

Sobald det Text der dritten den oben beschriebenen Abstand zum Seitenende erreicht, wird durch die Methode `accept_page_break` ein Seitenumbruch ausgelöst und die aktive Spalte sowie Schreibposition zurückgesetzt.

2.2.5 Lektion 5 - Tabellen erstellen

In dieser Lektion zeigen wir, wie man auf einfache Weise Tabellen erstellen kann.

Der Code wird drei verschiedene Tabellen erstellen, um zu zeigen, welche Effekte wir mit einigen einfachen Anpassungen erzielen können.

```
import csv
from fpdf import FPDF


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

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

    def colored_table(self, headings, rows, col_widths=(42, 39, 35, 42)):
        # Colors, line width and bold font:
        self.set_fill_color(255, 100, 0)
        self.set_text_color(255)
        self.set_draw_color(255, 0, 0)
        self.set_line_width(0.3)
        self.set_font(style="B")
        for col_width, heading in zip(col_widths, headings):
            self.cell(col_width, 7, heading, 1, 0, "C", True)
        self.ln()
        # Color and font restoration:
        self.set_fill_color(224, 235, 255)
        self.set_text_color(0)
        self.set_font()
        fill = False
        for row in rows:
            self.cell(col_widths[0], 6, row[0], "LR", 0, "L", fill)
```

```python
                self.cell(col_widths[1], 6, row[1], "LR", 0, "L", fill)
                self.cell(col_widths[2], 6, row[2], "LR", 0, "R", fill)
                self.cell(col_widths[3], 6, row[3], "LR", 0, "R", fill)
                self.ln()
                fill = not fill
            self.cell(sum(col_widths), 0, "", "T")


def load_data_from_csv(csv_filepath):
    headings, rows = [], []
    with open(csv_filepath, encoding="utf8") as csv_file:
        for row in csv.reader(csv_file, delimiter=","):
            if not headings:  # extracting column names from first row:
                headings = row
            else:
                rows.append(row)
    return headings, rows


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

Erzeugtes PDF - Länder

Da eine Tabelle lediglich eine Sammlung von Zellen darstellt, ist es naheliegend, eine Tabelle aus den bereits bekannten Zellen aufzubauen.

Das erste Beispiel wird auf die einfachste Art und Weise realisiert. Einfach gerahmte Zellen, die alle die gleiche Größe haben und linksbündig ausgerichtet sind. Das Ergebnis ist rudimentär, aber sehr schnell zu erzielen.

Die zweite Tabelle bringt einige Verbesserungen: Jede Spalte hat ihre eigene Breite, die Überschriften sind zentriert und die Zahlen rechtsbündig ausgerichtet. Außerdem wurden die horizontalen Linien entfernt. Dies geschieht mit Hilfe des Randparameters der Methode `cell()`, der angibt, welche Seiten der Zelle gezeichnet werden müssen. Im Beispiel wählen wir die linke (L) und die rechte (R) Seite. Jetzt muss nur noch das Problem der horizontalen Linie zum Abschluss der Tabelle gelöst werden. Es gibt zwei Möglichkeiten, es zu lösen: In der Schleife prüfen, ob wir uns in der letzten Zeile befinden und dann "LRB" als Rahmenparameter übergeben oder, wie hier geschehen, eine abschließende Zelle separat nach dem Durchlaufen der Schleife einfügen.

Die dritte Tabelle der zweiten sehr ähnlich, verwendet aber zusätzlich Farben. Füllung, Text und Linienfarben werden einfach mit den entsprechenden Methoden gesetzt. Eine wechselnde Färbung der Zeilen wird durch die abwechselnde Verwendung transparenter und gefüllter Zellen erreicht.

2.2.6 Lektion 6 - Links erstellen und Textstile mischen

In dieser Lektion werden verschiedene Möglichkeiten der Erstellung interner und externer Links beschrieben.

Es wird auch gezeigt, wie man verschiedene Textstile (fett, kursiv, unterstrichen) innerhalb eines Textes verwenden kann.

```python
from fpdf import FPDF, HTMLMixin


class MyFPDF(FPDF, HTMLMixin):
    pass


pdf = MyFPDF()

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

# Second page:
pdf.add_page()
pdf.set_link(link)
pdf.image(
    "../docs/fpdf2-logo.png", 10, 10, 50, 0, "", "https://pyfpdf.github.io/fpdf2/"
```

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

Erzeugtes PDF - fpdf2-logo

Die hier gezeigte neue Methode zur Einbindung von Text lautet `write()`. Sie ähnelt der bereits bekannten `multi_cell()`. Die wichtigsten Unterschiede sind:

- Das Ende der Zeile befindet sich am rechten Rand und die nächste Zeile beginnt am linken Rand.
- Die aktuelle Position wird an das Textende gesetzt.

Die Methode ermöglicht es uns somit, zuerst einen Textabschnitt zu schreiben, dann den Schriftstil zu ändern und genau an der Stelle fortzufahren, an der wir aufgehört haben. Der größte Nachteil ist jedoch, dass die von `multi_cell()` bekannte Möglichkeit zur Festlegung der Textausrichtung fehlt.

Auf der ersten Seite des Beispiels nutzen wir `write()`. Der Anfang des Satzes wird in "normalem" Stil geschrieben, dann mit der Methode `set_font()` auf Unterstreichung umgestellt und der Satz beendet.

Um einen internen Link hinzuzufügen, der auf die zweite Seite verweist, nutzen wir die Methode `add_link()`, die einen anklickbaren Bereich erzeugt, den wir "link" nennen und der auf eine andere Stelle innerhalb des Dokuments verweist. Auf der zweiten Seite verwenden wir `set_link()`, um den Zielbereich für den soeben erstellten Link zu definieren.

Um einen externen Link mit Hilfe eines Bildes zu erstellen, verwenden wir `image()`. Es besteht die Möglichkeit, der Methode ein Linkziel als eines ihrer Argumente zu übergeben. Der Link kann sowohl einer interner als auch ein externer sein.

Eine weitere Möglichkeit, den Schriftstil zu ändern und Links hinzuzufügen, stellt die Verwendung der Methode `write_html()` dar. Sie ist ein HTML-Parser, der das Hinzufügen von Text, Änderung des Schriftstils und Erstellen von Links mittels HTML ermöglicht.

2.3 Tutorial en español

Los diferentes ejemplos muestran rápidamente como usar fpdf2. Encontrará todas las características principales explicadas.

- Ejemplo básico
- Encabezado, pie de página, salto de página e imagen
- Saltos de línea y colores

2.3.1 Ejemplo básico

Empecemos con el ejemplo clásico:

```python
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")
```

Demo

Luego de incluir la biblioteca, creamos un objeto FPDF. El constructor FPDF es usado aquí con los valores predeterminados: páginas en A4 portrait -vertical- y la unidad de medida en milímetros. Podría haberlos especificado explícitamente:

```python
pdf=FPDF('P', 'mm', 'A4')
```

Es posible usar landscape -horizontal- (L), otros formatos de página (como Letter -carta- y Legal -oficio-) y unidad de medida (pt, cm, in).

Por el momento no hay una página, entonces tenemos que agregar una con add_page. El origen es la esquina superior-izquierda y la posición actual está ubicada a 1 cm de los bordes; los márgenes pueden ser cambiados con set_margins.

Antes de que podamos imprimir texto, es obligatorio seleccionar una fuente con set_font, de lo contrario, el documento será inválido. Elegimos helvetica bold 16:

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

Podríamos haber especificado italic -cursiva- con I, underline -subrayado- con U o fuente regular con string vacío (o cualquier combinación). Notar que el tamaño de la fuente es dado en puntos, no milímetros (u otra unidad de medida del usuario); ésta es la única excepción. Las otras fuentes estándar son Times, Courier, Symbol y ZapfDingbats.

Podemos ahora imprimir una celda con cell. Una celda es un área rectangular, posiblemente enmarcada, que contiene algún texto. Se imprime en la posición actual. Especificamos sus dimensiones, su texto (centrado o alineado), si los bordes deberían ser dibujados, y donde la posición actual se mueve después (a la derecha, abajo o al principio de la próxima linea). Para agregar un marco, haremos:

```python
pdf.cell(40, 10, 'Hola mundo !', 1)
```

Para agregar una nueva celda próxima a ella, con texto centrado y luego ir a la siguiente línea, haríamos:

```python
pdf.cell(60, 10, 'Hecho con FPDF.', new_x=XPos.LMARGIN, new_y=YPos.NEXT, align='C')
```

Nota: el salto de línea puede hacerse también con ln. Este método permite especificar adicionalmente la altura del salto.

Finalmente, el documento es cerrado y enviado al explorador con output. Podemos haberlo grabado a un fichero al pasarle el nombre de archivo.

Precaución: en caso cuando el PDF es enviado al explorador, nada más debe ser enviado a la salida, ni antes ni después (el mínimo caracter importa).

2.3.2 Encabezado, pie de página, salto de página e imagen

Aquí hay un ejemplo de dos páginas con encabezado, pie y logo:

```python
from fpdf import FPDF


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

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


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

Demo

Este ejemplo hace uso de métodos header y footer para procesar el encabezado y pie de página. Son llamados automáticamente. Ya existen en la clase FPDF pero no hacen nada por sí solos, por lo tanto tenemos que extender la clase y sobreescribirlos.

El logo es impreso con el método image especificando su esquina superior izquierda y su ancho. La altura es calculada automáticamente para respetar las proporciones de la imagen.

Para imprimir el número de página, un valor nulo es pasado como ancho de celda. Significa que la celda deberá ser extendida hasta el margen derecho de la página; es útil centrar texto. El número de página actual es devuelto por el método page_no; y para el número total de páginas, éste será obtenido mediante el valor especial {nb} que será sustituido al cerrar el documento. Notar el uso del método set_y que permite establecer la posición en una ubicación absoluta en la página, empezando desde arriba hacia abajo.

Otra característica interesante es usada aquí: el salto de página automático. Tan pronto una celda cruza el límite de una página (por defecto a 2 centímetros desde abajo), un salto es realizado y la fuente es restaurada. Aunque el encabezado y pie de página tienen su propia fuente (helvetica), el cuerpo continúa en Times. Este mecanismo de restauración automática también se aplica a los colores y el ancho de la línea. El límite que dispara los saltos de página puede establecerse con set_auto_page_break.

2.3.3 Saltos de línea y colores

Continuemos con un ejemplo que imprime párrafos justificados. También ilustra el uso de colores.

```python
from fpdf import FPDF


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

```
        # Performing a line break:
        self.ln(10)

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

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

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

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


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

Demo

El método get_string_width permite determinar la longitud de una cadena en la fuente actual, usado aquí para calcular la posición y el ancho del marco que rodea al título. Los colores son establecidos (vía set_draw_color, set_fill_color y set_text_color) y el grosor de la línea es establecido a 1 mm (contra 0.2 por defecto) con set_line_width. Finalmente, emitimos la celda (el último parámetro es True para indicar que el fondo debe ser rellenado).

El método usado para imprimir párrafos es multi_cell. Cada vez que una línea alcanza el extremo derecho de la celda o un caracter de retorno de línea, un salto de línea es emitido y una nueva celda es automáticamente creada bajo la actual. El texto es justificado por defecto.

Dos propiedades del documento son definidas: el título (set_title) y el autor (set_author). Las propiedades pueden ser vistas de dos formas. La primera es abrir el documento directamente con Acrobat Reader, ir al menú Archivo y elegir la opción Propiedades del Documento. La segunda, también disponible en el plug-in, es hacer clic izquierdo y seleccionar Propiedades del documento (Document Properties).

2.4 Tutorial

Documentation complète des méthodes : `fpdf.FPDF` API doc

2.4.1 Tuto 1 - Exemple minimal

Commençons par un exemple classique :

```python
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")
```

PDF généré

Après avoir inclu la librairie, on créé un objet `FPDF`. Le constructeur FPDF est utilisé avec ses valeurs par défaut : les pages sont en format portrait A4 et l'unité de mesure est le millimètre. Cela peut également être spéficié de cette manière :

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

Il est possible de créer un PDF en format paysage (`L`) ou encore d'utiliser d'autres formats (par exemple `Letter` et `Legal`) et unités de mesure (`pt`, `cm`, `in`).

Il n'y a pas encore de page, il faut donc en créer une avec add_page. Le coin en haut à gauche correspond à l'origine, et le curseur (c'est-à-dire la position actuelle où l'on va afficher un élément) est placé par défaut à 1 cm des bords; les marges peuvent être modifiées avec set_margins.

Avant de pouvoir afficher du texte, il faut obligatoirement choisir une police de caractères avec set_font. Choisissons Helvetica bold 16:

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

On aurait pu spécifier une police en italique avec `I`; soulignée avec `U` ou une police normale avec une chaine de caractères vide. Il est aussi possible de combiner les effets en combinant les caractères. Notez que la taille des caractères est à spécifier en points (pts), pas en millimètres (ou tout autre unité); c'est la seule exception. Les autres polices fournies par défaut sont `Times`, `Courier`, `Symbol` et `ZapfDingbats`.

On peut maintenant afficher une cellule avec cell. Une cellule est une zone rectangulaire, avec ou sans cadre, qui contient du texte. Elle est affichée à la position actuelle du curseur. On spécifie ses dimensions, le texte (centré ou aligné), s'il y a une bordure ou non, ainsi que la position du curseur après avoir affiché la cellule (s'il se déplace à droite, vers le bas ou au début de la ligne suivante). Pour ajouter un cadre, on utilise ceci :

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

Pour ajouter une nouvelle cellule avec un texte centré, et déplacer le curseur à la ligne suivante on utilise cela :

```python
pdf.cell(60, 10, 'Powered by FPDF.', new_x=XPos.LMARGIN, new_y=YPos.NEXT, align='C')
```

Remarque : le saut de ligne peut aussi être fait avec ln. Cette méthode permet de spécifier la hauteur du saut.

Enfin, le document est sauvegardé à l'endroit spécifié en utilisant output. Sans aucun paramètre, `output()` retourne le buffer `bytearray` du PDF.

2.4.2 Tuto 2 - En-tête, bas de page, saut de page et image

Voici un exemple contenant deux pages avec un en-tête, un bas de page et un logo :

```python
from fpdf import FPDF


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

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


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

PDF généré

Cet exemple utilise les méthodes header et footer pour générer des en-têtes et des bas de page. Elles sont appelées automatiquement. Elles existent déjà dans la classe FPDF mais elles ne font rien, il faut donc les redéfinir dans une classe fille.

Le logo est affiché avec la méthode image en spécifiant la position du coin supérieur gauche et la largeur de l'image. La hauteur est calculée automatiquement pour garder les proportions de l'image.

Pour centrer le numéro de page dans le bas de page, il faut passer la valeur nulle à la place de la largeur de la cellule. Cela fait prendre toute la largeur de la page à la cellule, ce qui permet de centrer le texte. Le numéro de page actuel est obtenu avec la méthode page_no; le nombre total de pages est obtenu avec la variable `{nb}` qui prend sa valeur quand le document est fermé (la méthode alias_nb_pages permet de définir un autre nom de variable pour cette valeur). La méthode set_y permet de spécifier une position dans la page relative au haut ou pas de page.

Une autre fonctionnalité intéressante est utilisée ici : les sauts de page automatiques. Si une cellule dépasse la limite du contenu de la page (par défaut à 2 centimètres du bas), un saut de page est inséré à la place et la police de caractères est restaurée. C'est-à-dire, bien que l'en-tête et le bas de page utilisent la police (`helvetica`), le corps du texte garde la police `Times` . Ce mécanisme de restauration automatique s'applique également à la couleur et l'épaisseur des lignes. La limite du contenu qui déclenche le saut de page peut être spécifiée avec set_auto_page_break.

2.4.3 Tuto 3 - Saut de ligne et couleur

Continuons avec un exemple qui affiche des paragraphes avec du texte justifié. Cet exemple montre également l'utilisation de couleurs.

```python
from fpdf import FPDF


class PDF(FPDF):
    def header(self):
        # Setting font: helvetica bold 15
        self.set_font("helvetica", "B", 15)
```

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

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

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

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

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


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

PDF généré

Texte de Jules Verne

La méthode get_string_width permet de déterminer la largeur d'un texte utilisant la police actuelle, ce qui permet de calculer la position et la largeur du cadre autour du titre. Ensuite les couleurs sont spécifiées (avec set_draw_color, set_fill_color et set_text_color) et on spécifie l'épaisseur de la bordure du cadre à 1 mm (contre 0.2 par défaut) avec set_line_width. Enfin, on affiche la cellule (le dernier paramètre "true" indique que le fond doit être rempli).

La méthode multi_cell est utilisée pour afficher les paragraphes. Chaque fois qu'une ligne atteint le bord d'une cellule ou qu'un caractère de retour à la ligne est présent, un saut de ligne est inséré et une nouvelle cellule est créée automatiquement sous la cellule actuelle. Le texte est justifié par défaut.

Deux propriétés sont définies pour le document : le titre (set_title) et l'auteur (set_author). Les propriétés peuvent être trouvées en ouvrant le document PDF avec Acrobat Reader. Elles sont alors visibles dans le menu Fichier -> Propriétés du document.

2.4.4 Tuto 4 - Colonnes multiples

En cours de traduction.

2.4.5 Tuto 5 - Créer des tables

En cours de traduction.

2.4.6 Tuto 6 - Créer des liens et mélanger différents styles de textes

En cours de traduction.

2.5 Tutorial

Documentazione completa dei metodi: `fpdf.FPDF` API doc

- Tutorial
- Tuto 1 - Esempio base
- Tuto 2 - Intestazione, piè di pagina, interruzione di pagina ed immagini
- Tuto 3 - Interruzioni di riga e colori
- Tuto 4 - Colonne multiple
- Tuto 5 - Creare tabelle
- Tuto 6 - Creare link e mescolare stili di testo

2.5.1 Tuto 1 - Esempio base

Iniziamo con un esempio comune:

```
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")
```

Risultato PDF

Dopo aver incluso la libreria, creiamo un oggetto `FPDF`. Così facendo il costruttore FPDF viene utilizzato con i suoi valori di default: le pagine sono in A4 verticale e l'unità di misura è millimetri. Avremmo potuto specificarle esplicitamente facendo:

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

È possibile impostare il PDF in modalità orizzontale (`L`) o utilizzare altri formati (come `Letter` e `Legal`) e unità di misura (`pt` , `cm` , `in`).

Non esiste una pagina al momento, quindi dobbiamo aggiungerne una con add_page. L'origine è in alto a sinistra e la posizione corrente è a 1cm dai bordi; i margini possono essere cambiati con set_margins.

Prima di poter stampare del testo, è obbligatorio selezionare un font con set_font, altrimenti il documento risulterebbe non valido. Scegliamo Helvetica bold 16:

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

Avremmo potuto scegliere il corsivo con `I` , sottolineato con `U` o un font regolare lasciando la stringa vuota (o ogni combinazione). Notare che la dimensione dei caratteri è specificata in punti, non millimetri (o altre unità di misura); questa è l'unica eccezione. Gli altri font disponibili sono `Times` , `Courier` , `Symbol` and `ZapfDingbats` .

Adesso possiamo disegnare una cella con cell. Una cella è un'area rettangolare, in caso con bordo, che contiene del testo. È disegnata nella attuale posizione. Specifichiamo le sue dimensioni, il suo testo (centrato o allineato), se i bordi devono essere mostrati, e dove verrà spostata la posizione quando avremo finito (a destra, sotto, o all'inizio della riga successiva). Faremmo così:

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

Per aggiungere una nuova cella di fianco alla precedente con testo centrato e poi spostarci alla riga successiva, faremmo:

```
pdf.cell(60, 10, 'Powered by FPDF.', new_x=XPos.LMARGIN, new_y=YPos.NEXT, align='C')
```

NB: si può andare a capo anche con ln. Questo metodo permette di specificare l'altezza dello spazio.

In fine, il documento è chiuso e salvato nella destinazione fornita attraverso output. Senza alcun parametro, `output()` ritorna il PDF in un buffer `bytearray`.

2.5.2 Tuto 2 - Intestazione, piè di pagina, interruzione di pagina ed immagini

Ecco un esempio composto da due pagine con intestazione, piè di pagina e logo:

```python
from fpdf import FPDF


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

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


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

Risultato PDF

Questo esempio sfrutta i metodi header e footer per processare intestazioni e piè di pagina. Vengono chiamati automaticamente. Esistono nella classe FPDF ma non eseguono operazioni, quindi è necessario estendere la classe e sovrascriverli.

Il logo è stampato con il metodo image specificando la posizione del suo angolo in alto a sinistra e la sua larghezza. L'altezza è calcolata automaticamente per rispettare le proporzioni dell'immagine.

Per stampare il numero della pagina, un valore nullo può essere passato come larghezza della cella. Significa che la cella "crescerà" fino al margine destro della pagina; è utile per centrare il testo. Il numero di pagina è ritornato da page_no; mentre per il numero totale di pagine, si ottiene attraverso il valore speciale `{nb}` che verrà sostituito quando le pagine saranno generate. Importante menzionare il metodo set_y che permette di selezionare una posizione assoluta all'interno della pagina, incominciando dall'alto o dal basso.

Un'altra feature interessante: l'interruzione di pagina automatica. Non appena una cella dovesse superare il limite nella pagina (a 2 centimetri dal fondo di default), ci sarebbe un'interruzione e un reset del font. Nonostante l'intestazione e il piè di pagina scelgano il proprio font (`helvetica`), il contenuto continua in `Times`. Questo meccanismo di reset automatico si applica anche ai colori e allo spessore della linea. Il limite può essere scelto con set_auto_page_break.

2.5.3 Tuto 3 - Interruzioni di riga e colori

Continuiamo con un esempio che stampa paragrafi giustificati. Mostreremo anche l'utilizzo dei colori.

```python
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)
```

```python
        self.set_text_color(220, 50, 50)
        # Setting thickness of the frame (1 mm)
        self.set_line_width(1)
        # Printing title:
        self.cell(width, 9, self.title, 1, 1, "C", True)
        # Performing a line break:
        self.ln(10)

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

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

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

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


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

Risultato PDF

Testo Jules Verne

Il metodo get_string_width permette di determinare la lunghezza di una stringa nel font selezionato, e viene utilizzato per calcolare la posizione e la larghezza della cornice intorno al titolo. Successivamente selezioniamo i colori (utilizzando set_draw_color, set_fill_color e set_text_color) e aumentiamo la larghezza della linea a 1mm (invece dei 0.2 di default) con set_line_width. In fine, stampiamo la cella (l'ultimo parametro a true indica che lo sfondo dovrà essere riempito).

Il metodo utilizzato per stampare i paragrafi è multi_cell. Ogni volta che una linea raggiunge l'estremità destra della cella o c'è un carattere carriage return, avremo un'interruzione di linea e una nuova cella verrà automaticamente creata. Il testo è giustificato di default.

Due proprietà del documento vengono definite: il titolo (set_title) e l'autore (set_author). Le proprietà possono essere controllate in due modi. Il primo è aprire direttamente il documento con Acrobat Reader, cliccare sul menù File e scegliere l'opzione Proprietà del documento. la seconda, è di cliccare con il tasto destro e scegliere Proprietà del documento.

2.5.4 Tuto 4 - Colonne multiple

Questo esempio è una variante del precedente, mostra come disporre il test attraverso colonne multiple.

```python
from fpdf import FPDF


class PDF(FPDF):
```

```python
    def __init__(self):
        super().__init__()
        self.col = 0  # Current column
        self.y0 = 0  # Ordinate of column start

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

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

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

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

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

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

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


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

Risultato PDF

Testo Jules Verne

La differenza dal precedente tutoria è l'utilizzo dei metodi accept_page_break e set_col.

Usando accept_page_break, una volta che la cella oltrepasserà il limite basso della pagina, il numero della colonna corrente verrà controllato. Se sarà meno di 2 (abbiamo deciso di dividere la pagina in 3 colonne) chiamerà il metodo set_cal, incrementando il numero della colonna e modificando la posizione della prossima colonna in modo e il testo continui lì.

Una volta che il limite inferiore della terza colonna sarà raggiunto, accept_page_break resetterà e andrà alla prima colonna provocando una interruzione di pagina.

2.5.5 Tuto 5 - Creare tabelle

Questo tutoria spiegherà come creare facilmente tabelle.

Creeremo tre diverse tabelle per spiegare cosa si può ottenere con piccolo cambiamenti.

```python
import csv
from fpdf import FPDF


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

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

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


def load_data_from_csv(csv_filepath):
    headings, rows = [], []
    with open(csv_filepath, encoding="utf8") as csv_file:
        for row in csv.reader(csv_file, delimiter=","):
            if not headings:  # extracting column names from first row:
                headings = row
            else:
                rows.append(row)
    return headings, rows


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

Risultato PDF - Testo delle nazioni

Dato che una tabella è un insieme di celle, viene natura crearne una partendo da loro.

Il primo esempio è la via più elementare: semplici celle con cornice, tutte della stessa dimensione e allineate a sinistra. Il risultato è rudimentale ma molto veloce da ottenere.

La seconda tabella contiene dei miglioramenti: ogni colonna ha la propria larghezza, i titoli sono centrati e i numeri allineati a destra. Inoltre, le linee orizzontale sono state rimosse. Questo è stato possibile grazie al parametro border del metodo Cell(), che specifica quali lati della cella saranno disegnati. In questo caso vogliamo il sinistro (L) e il destro (R). Rimane il problema delle linee orizzontali. Ci sono due possibilità per risolverlo: controllare di essere nell'ultimo giro del ciclo, nel qual caso utilizziamo LRB per il parametro border; oppure, come fatto in questo esempio, aggiungiamo una linea dopo il completamento del ciclo.

La terza tabella è molto simile alla seconda, ma utilizza i colori. Il colore di sfondo, testo e linee sono semplicemente specificati. L'alternanza dei colori delle righe è ottenuta utilizzando celle con sfondo colorato e trasparente alternativamente.

2.5.6 Tuto 6 - Creare link e mescolare stili di testo

Questo tutorial spiegherà molti modi di inserire link interni al pdf, e come inserirne a sorgenti esterne.

Saranno mostrati anche molti modi di utilizzare diversi stili di testo (grassetto, corsivo e sottolineato) nello stesso testo.

```python
from fpdf import FPDF, HTMLMixin


class MyFPDF(FPDF, HTMLMixin):
    pass


pdf = MyFPDF()

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

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

Risultato PDF - fpdf2-logo

Il nuovo metodo qui utilizzato per stampare testo è write() . È molto simile a multi_cell() , ma con delle differenze:

- La fine della linea è al margine destro e la linea successiva inizia al margine sinistro.
- La posizione attuale si sposta alla fine del testo stampato.

Il metodo quindi ci permette di scrivere un blocco di testo, cambiare lo stile del testo, e continuare a scrivere esattamente da dove eravamo rimasti. D'altro canto, il suo peggior svantaggio è che non possiamo giustificare il testo come con multi_cell() method.

Nella prima pagina dell'esempio, abbiamo usato write() per questo scopo. L'inizio della frase è scritta in font normale, poi utilizzando set_font() siamo passati al sottolineato e abbiamo finito la frase.

Per aggiungere un link interno che puntasse alla seconda pagina, abbiamo utilizzato add_link() che crea un area cliccabile che abbiamo chiamato "link" che redirige ad un altro punto del documento. Nella seconda pagina abbiamo usato set_link() per definire un'area di destinazione per il link creato in precedenza.

Per creare un link esterno utilizzando un'immagine, abbiamo usato image() . Il metodo ha l'opzione di passare un link come argomento. Il link può essere sia interno che esterno.

In alternativa, un'altra opzione per cambiare lo stile e aggiungere link è di utilizzare `write_html()`. È un parser hrml che permette di aggiungere testo, cambiare stile e aggiungere link utilizzando html.

2.6 Tutorial

Methods full documentation: `fpdf.FPDF` API doc

2.6.1 Tuto 1 - Exemplo Mínimo

Vamos começar com um exemplo clássico:

```python
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")
```

PDF resultante

Após incluirmos o ficheiro da biblioteca, criamos um objeto `FPDF`. O FPDF construtor é construído com os seguintes parâmetros por omissão: Páginas são em formato A4 vertical e a unidade de medida é o milímetro. Pode ser especificado explicitamente através de:

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

É possível colocar o PDF em modo horizontal (`L`) ou em outros formatos de página (como `Letter` e `Legal`) e em outras unidades de medida (`pt` , `cm` , `in`).

Neste momento, não há nenhuma página, então temos que adicionar uma com add_page. A origem está no canto superior esquerdo e a posição atual é, por padrão, colocada a 1 cm das bordas; as margens podem ser alteradas com set_margins.

Antes de imprimirmos o texto, é obrigatório selecionar uma fonte com set_font, caso contrário, o documento será inválido. Nós escolhemos Helvetica bold 16:

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

Podemos formatar em itálico com `I` , sublinhar com `U` ou uma fonte normal com uma string vazia (ou qualquer combinação). Observe que o tamanho da fonte é fornecido em pontos, não milímetros (ou outra unidade do utilizador); esta é a única exceção. As outras fontes integradas são `Times` , `Courier` , `Symbol` e `ZapfDingbats` .

Agora podemos imprimir uma célula com cell. Uma célula é uma área retangular, possivelmente emoldurada, que contém algum texto. É renderizado na posição atual. Nós especificamos as suas dimensões, o seu texto (centrado ou alinhado), se as bordas devem ser desenhadas, e para onde a posição atual se deve mover depois desta alteração (para a direita, abaixo ou no início da próxima linha). Para adicionar uma moldura, temos de fazer o seguinte:

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

Para adicionar uma nova célula ao lado desta, com texto centralizado e ir para a próxima linha, teríamos de fazer:

```python
pdf.cell(60, 10, 'Powered by FPDF.', new_x=XPos.LMARGIN, new_y=YPos.NEXT, align='C')
```

Nota: a quebra de linha também pode ser feita com ln. Esse método permite especificar, adicionalmente, a altura da quebra.

Finalmente, o documento é fechado e guardado no caminho do arquivo fornecido utilizando output. Sem termos qualquer parâmetro fornecido, `output ()` retorna o buffer PDF `bytearray`.

2.6.2 Tuto 2 - Cabeçalho, rodapé, quebra de página e imagem

Aqui temos um exemplo de duas páginas com cabeçalho, rodapé e logótipo:

```python
from fpdf import FPDF


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

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


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

PDF resultante

Este exemplo usa os header e o footer para processar cabeçalhos e rodapés de página. Estes são chamados automaticamente. Eles já existem na classe FPDF, mas não fazem nada, portanto, temos que os estender a classe e substituí-los.

O logótipo é impresso utilizando o método image, especificando o seu canto superior esquerdo e sua largura. A altura é calculada automaticamente para respeitar as proporções da imagem.

Para imprimir o número da página, um valor nulo é passado como a largura da célula. Isso significa que a célula deve se estender até a margem direita da página; é útil para centralizar texto. O número da página atual é retornado pelo método page_no; quanto ao número total de páginas, é obtido por meio do valor especial `{nb}` que será substituído quando se fecha o documento. Observe que o uso do método set_y permite definir a posição em um local absoluto da página, começando do início ou do fim.

Outro recurso interessante que se usa aqui é a quebra de página automática. Desde do momento em que uma célula cruza o limite da página (a 2 centímetros da parte inferior por padrão), uma pausa é executada e a fonte restaurada. Embora o cabeçalho e rodapés selecionam a sua própria fonte (`helvetica`), o corpo continua com `Times`. Este mecanismo de restauração automática também se aplica a cores e largura de linha. O limite que dispara quebras de página pode ser definido com set_auto_page_break.

2.6.3 Tuto 3 - Quebras de linha e cores

Vamos continuar com um exemplo que imprime parágrafos justificados e o uso de cores.

```python
from fpdf import FPDF


class PDF(FPDF):
    def header(self):
        # Setting font: helvetica bold 15
        self.set_font("helvetica", "B", 15)
        # Calculating width of title and setting cursor position:
        width = self.get_string_width(self.title) + 6
        self.set_x((210 - width) / 2)
        # Setting colors for frame, background and text:
        self.set_draw_color(0, 80, 180)
        self.set_fill_color(230, 230, 0)
        self.set_text_color(220, 50, 50)
```

```
        # Setting thickness of the frame (1 mm)
        self.set_line_width(1)
        # Printing title:
        self.cell(width, 9, self.title, 1, 1, "C", True)
        # Performing a line break:
        self.ln(10)

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

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

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

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


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

PDF resultante

Texto de Júlio Verne

O método get_string_width permite determinar o comprimento de uma string na fonte atual, e que é usada aqui para calcular a posição e a largura do quadro ao redor do título. Em seguida, as cores são definidas (via set_draw_color, set_fill_color e set_text_color) e a espessura da linha é definida como 1 mm (contra 0,2 por padrão) com set_line_width. Finalmente, produzimos a célula (se o último parâmetro for verdadeiro, indica que o plano de fundo deve ser preenchido).

O método usado para imprimir os parágrafos é multi_cell. Cada vez que uma linha atinge a extremidade direita da célula ou um código de fim de linha é encontrado, uma quebra de linha é emitida e uma nova célula é criada automaticamente sob a atual. O texto é justificado por padrão.

Duas propriedades do documento são definidas: o título (set_title) e o autor (set_author). As propriedades podem ser visualizadas de duas maneiras: A primeira é abrir o documento diretamente com o Acrobat Reader, vá para o menu Arquivo e escolha a opção Propriedades do documento. O segundo, também disponível no plug-in, é clicar com o botão direito e selecionar Propriedades do documento.

2.6.4 Tuto 4 - Multi Colunas

Este exemplo é uma variante do anterior, mostrando como colocar o texto em várias colunas.

```
from fpdf import FPDF


class PDF(FPDF):
    def __init__(self):
```

```python
        super().__init__()
        self.col = 0  # Current column
        self.y0 = 0  # Ordinate of column start

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

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

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

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

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

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

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


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

PDF resultante

Texto de Júlio Verne

A principal diferença em relação ao tutorial anterior é o uso do accept_page_break e os métodos set_col.

Usando o método accept_page_break, quando a célula ultrapassar o limite inferior da página, ela verificará o número da coluna atual. Se isso for menor que 2 (optamos por dividir a página em três colunas), chamando o método set_col, aumentando o número da coluna e alterando a posição da próxima coluna para que o texto continue aí.

Quando o limite inferior da terceira coluna é alcançado, o método accept_page_break será redefinido e vai voltar para a primeira coluna e adicionar uma quebra de página.

2.6.5 Tuto 5 - Criar Tabelas

Este tutorial irá explicar como criar tabelas facilmente.

O código seguinte cria três tabelas diferentes para explicar o que pode ser alcançado com alguns ajustes simples.

```python
import csv
from fpdf import FPDF


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

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

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


def load_data_from_csv(csv_filepath):
    headings, rows = [], []
    with open(csv_filepath, encoding="utf8") as csv_file:
        for row in csv.reader(csv_file, delimiter=","):
            if not headings:  # extracting column names from first row:
                headings = row
            else:
                rows.append(row)
    return headings, rows


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

PDF resultante - Texto dos países

Uma vez que uma tabela é apenas uma coleção de células, é natural construir uma a partir delas.

O primeiro exemplo é obtido da maneira mais básica possível: moldura simples células, todas do mesmo tamanho e alinhadas à esquerda. O resultado é rudimentar, mas muito rápido de obter.

A segunda tabela traz algumas melhorias: cada coluna tem sua largura própria, os títulos estão centrados e as figuras alinhadas à direita. Além disso, as linhas horizontais foram removidas. Isto é feito por meio do parâmetro border do método Cell(), que especifica quais lados da célula devem ser desenhados. Aqui nós queremos os esquerdo (L) e direito (R). Agora apenas o problema da linha horizontal para terminar a mesa permanece. Existem duas possibilidades para resolvê-lo: verificar para a última linha do loop, caso este em que usamos LRB para o parâmetro da borda; ou, como foi feito aqui, adicione a linha assim que o loop terminar.

A terceira tabela é semelhante à segunda, mas usa cores. Preenchimento, texto e as cores das linhas são simplesmente especificadas. Coloração alternativa para linhas é obtida usando células alternativamente transparentes e preenchidas.

2.6.6 Tuto 6 - Criar links e misturar estilos de texto

Este tutorial irá explicar várias maneiras de inserir links dentro de um documento PDF, bem como adicionar links para fontes externas.

Também mostrará várias maneiras de usar diferentes estilos de texto, (negrito, itálico, sublinhado) no mesmo texto.

```python
from fpdf import FPDF, HTMLMixin


class MyFPDF(FPDF, HTMLMixin):
    pass


pdf = MyFPDF()

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

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

PDF resultante - fpdf2-logo

O novo método mostrado aqui para imprimir texto é write(). É muito parecido com multi_cell (), sendo as principais diferenças:

- O fim da linha está na margem direita e a próxima linha começa na margem esquerda.
- A posição atual move-se para o final do texto.

O método, portanto, nos permite escrever um pedaço de texto, alterar o estilo da fonte, e continuar do ponto exato em que paramos. Por outro lado, a sua principal desvantagem é que não podemos justificar o texto como nós fazemos com o método [multi_cell()](https://pyfpdf.github.io/fpdf2/fpdf/fpdf.html#fpdf.fpdf.FPDF.multi_cell) .

Na primeira página do exemplo, usámos write() para este propósito. O início da frase está escrita no estilo de texto normal, depois usando o método set_font(), trocamos para sublinhado e acabámos a frase.

Para adicionar o link externo a apontar para a segunda página, nós usámos o método add_link(), que cria uma área clicável à qual demos o nome de "link" que direciona para outra parte do documento. Na segunda página, usámos set_link() para definir uma área de destino para o link que acabámos de criar.

Para criar o link externo usando uma imagem, usámos image(). O método tem a opção de passar um link como um dos seus argumentos. O link pode ser interno ou externo.

Como alternativa, outra opção para mudar o estilo da fonte e adicionar links é usar o método `write_html()`. É um "parser" que permite adicionar texto, mudar o estilo da fonte e adicionar links usando html.

2.7 Руководство

Полная документация по методам класса **FPDF**: `fpdf.FPDF` API doc

- Руководство
- Руководство 1 - Минимальный пример
- Руководство 2 - Верхний колонтитул, нижний колонтитул, разрыв страницы и картинка
- Руководство 3 - Переносы строк и цвета
- Руководство 4 - Несколько колонок
- Руководство 5 - Создание таблиц
- Руководство 6 - Создание ссылок и смешивание стилей текста

2.7.1 Руководство 1 - Минимальный пример

Начнём с классического примера:

```
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")
```

Итоговый PDF

После подключения библиотеки мы создаем объект `FPDF`. Здесь используется конструктор FPDF со значениями по умолчанию: страницы формата A4 портретные, единица измерения - миллиметр.

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

Можно установить PDF в альбомном режиме (`L`) или использовать другой формат страниц (например, `Letter` или `Legal`) и единицы измерения (`pt` , `cm` , `in`).

На данный момент страницы нет, поэтому мы должны добавить ее с помощью команды add_page. Начало страницы находится в левом верхнем углу, а текущая позиция по умолчанию располагается на расстоянии 1 см от границ; поля можно изменить с помощью команды set_margins.

Прежде чем мы сможем напечатать текст, обязательно нужно выбрать шрифт с помощью set_font, иначе документ будет недействительным. Мы выбираем Helvetica bold 16:

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

Мы можем указать курсив с помощью `I` , подчеркнутый шрифт с помощью `U` или обычный шрифт с помощью пустой строки (или использовать любую комбинацию). Обратите внимание, что размер шрифта задается в пунктах, а не в миллиметрах (или другой единице измерений); это единственное исключение. Другие встроенные шрифты: `Times` , `Courier` , `Symbol` и `ZapfDingbats` .

Теперь мы можем распечатать ячейку с помощью cell. Ячейка - это прямоугольная область, возможно, обрамленная рамкой, которая содержит некоторый текст. Она отображается в текущей позиции. Мы указываем ее размеры, текст (центрированный или выровненный), должны ли быть нарисованы рамки, и куда текущая позиция перемещается после нее (вправо, вниз или в начало следующей строки). Чтобы добавить рамку, мы сделаем следующее:

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

Чтобы добавить новую ячейку с центрированным текстом и перейти к следующей строке, мы сделаем следующее:

```
pdf.cell(60, 10, 'Powered by FPDF.', new_x=XPos.LMARGIN, new_y=YPos.NEXT, align='C')
```

Примечание: разрыв строки также можно сделать с помощью ln. Этот метод позволяет дополнительно указать высоту разрыва.

Наконец, документ закрывается и сохраняется по указанному пути к файлу с помощью функции output. Без указания параметров `output()` возвращает буфер PDF `bytearray`.

2.7.2 Руководство 2 - Верхний колонтитул, нижний колонтитул, разрыв страницы и картинка

Пример двух страниц с верхним и нижним колонтитулами и логотипом:

```python
from fpdf import FPDF


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

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


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

Итоговый PDF

В этом примере используются методы header и footer для обработки заголовков и колонтитулов страницы. Они вызываются автоматически. Они уже существуют в классе FPDF, но ничего не делают, поэтому мы должны расширить класс и переопределить их.

Логотип печатается методом image с указанием его левого верхнего угла и ширины. Высота вычисляется автоматически, чтобы соблюсти пропорции изображения.

Для печати номера страницы в качестве ширины ячейки передается нулевое значение. Это означает, что ячейка должна простираться до правого поля страницы; это удобно для центрирования текста. Номер текущей страницы возвращается методом page_no; что касается общего количества страниц, то оно получается с помощью специального значения `{nb}`, которое будет подставлено при закрытии документа. Обратите внимание на использование метода set_y, который позволяет установить позицию в абсолютном месте страницы, начиная сверху или снизу.

Здесь используется еще одна интересная функция: автоматический разрыв страницы. Как только ячейка пересекает границу страницы (по умолчанию 2 сантиметра от низа), происходит разрыв и шрифт восстанавливается. Хотя верхний и нижний колонтитулы выбирают свой собственный шрифт (`helvetica`), основная часть продолжает использовать `Times`. Этот механизм автоматического восстановления также применяется к цветам и ширине линий. Предел, который вызывает разрыв страницы, можно установить с помощью set_auto_page_break.

2.7.3 Руководство 3 - Переносы строк и цвета

Продолжим с примера, который печатает выровненные абзацы. Он также иллюстрирует использование цветов.

```python
from fpdf import FPDF


class PDF(FPDF):
    def header(self):
```

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

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

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

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

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


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

Итоговый PDF

Текст Жюля Верна

Метод get_string_width позволяет определить длину строки в текущем шрифте, которая используется здесь для расчета положения и ширины рамки, окружающей заголовок. Затем устанавливаются цвета (через set_draw_color, set_fill_color и set_text_color), а толщина линии устанавливается в 1 мм (против 0,2 по умолчанию) с помощью set_line_width. Наконец, мы выводим ячейку (последний параметр True указывает на то, что фон должен быть заполнен).

Для печати абзацев используется метод multi_cell. Каждый раз, когда строка достигает правого края ячейки или встречается символ возврата каретки, выдается разрыв строки и автоматически создается новая ячейка под текущей. По умолчанию текст выравнивается по ширине.

Определены два свойства документа: заголовок (set_title) и автор (set_author). Свойства можно просматривать двумя способами. Первый - открыть документ непосредственно с помощью Acrobat Reader, перейти в меню Файл и выбрать пункт Свойства документа. Второй, также доступный из плагина, - щелкнуть правой кнопкой мыши и выбрать пункт Свойства документа.

2.7.4 Руководство 4 - Несколько колонок

Этот пример является вариантом предыдущего и показывает, как расположить текст в нескольких колонках.

```python
from fpdf import FPDF


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

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

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

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

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

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

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

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


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

Итоговый PDF

Текст Жюля Верна

Ключевым отличием от предыдущего урока является использование методов accept_page_break и set_col.

С помощью метода accept_page_break, в тот момент, когда ячейка пересекает нижнюю границу страницы, проверяется номер текущей колонки. Если он меньше 2 (мы решили разделить страницу на три колонки), то будет вызван метод set_col, увеличивающий номер колонки и изменяющий положение следующей колонки, чтобы текст мог быть продолжен в ней.

Как только будет достигнута нижняя граница третьей колонки, метод accept_page_break произведет сброс и возврат к первой колонке и инициирует разрыв страницы.

2.7.5 Руковдство 5 - Создание таблиц

В этом уроке мы расскажем, как можно с легкостью создавать таблицы.

Код создаст три различные таблицы, чтобы объяснить, чего можно достичь с помощью нескольких простых настроек.

```python
import csv
from fpdf import FPDF


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

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

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


def load_data_from_csv(csv_filepath):
    headings, rows = [], []
    with open(csv_filepath, encoding="utf8") as csv_file:
        for row in csv.reader(csv_file, delimiter=","):
            if not headings:  # extracting column names from first row:
                headings = row
            else:
                rows.append(row)
    return headings, rows


col_names, data = load_data_from_csv("countries.txt")
pdf = PDF()
pdf.set_font("helvetica", size=14)
pdf.add_page()
```

```
pdf.basic_table(col_names, data)
pdf.add_page()
pdf.improved_table(col_names, data)
pdf.add_page()
pdf.colored_table(col_names, data)
pdf.output("tuto5.pdf")
```

Итоговый PDF - Список стран

Поскольку таблица - это просто набор ячеек, естественно построить таблицу из них.

Первый пример достигается самым простым способом: простые ячейки в рамке, все одинакового размера и выровненные по левому краю. Результат элементарен, но достигается очень быстро.

Вторая таблица имеет некоторые улучшения: каждый столбец имеет свою ширину, заголовки выровнены по центру, а цифры - по правому краю. Более того, горизонтальные линии были удалены. Это сделано с помощью параметра border метода Cell(), который указывает, какие стороны ячейки должны быть нарисованы. Здесь нам нужны левая (L) и правая (R). Теперь остается только проблема горизонтальной линии для завершения таблицы. Есть две возможности решить ее: проверить наличие последней строки в цикле, в этом случае мы используем LRB для параметра границы; или, как сделано здесь, добавить линию после завершения цикла.

Третья таблица похожа на вторую, но в ней используются цвета. Цвета заливки, текста и линий просто задаются. Альтернативная окраска строк достигается путем использования поочередно прозрачных и заполненных ячеек.

2.7.6 Руководство 6 - Создание ссылок и смешивание стилей текста

В этом уроке будет рассказано о нескольких способах вставки ссылок внутри pdf документа, а также о добавлении ссылок на внешние источники.

Также будет показано несколько способов использования различных стилей текста (жирный, курсив, подчеркивание) в одном и том же тексте.

```python
from fpdf import FPDF, HTMLMixin


class MyFPDF(FPDF, HTMLMixin):
    pass


pdf = MyFPDF()

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

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

Итоговый PDF - fpdf2-logo

Новый метод, показанный здесь для печати текста - это write(). Он очень похож на multi_cell(), основные отличия заключаются в следующем:

- Конец строки находится на правом поле, а следующая строка начинается на левом поле.

- Текущая позиция перемещается в конец текста.

Таким образом, этот метод позволяет нам написать фрагмент текста, изменить стиль шрифта и продолжить с того самого места, на котором мы остановились. С другой стороны, его главный недостаток заключается в том, что мы не можем выровнять текст, как это делается при использовании метода multi_cell().

На первой странице примера мы использовали для этой цели write(). Начало предложения написано текстом обычного стиля, затем, используя метод set_font(), мы переключились на подчеркивание и закончили предложение.

Для добавления внутренней ссылки, указывающей на вторую страницу, мы использовали метод add_link(), который создает кликабельную область, названную нами "link", которая ведет в другое место внутри документа. На второй странице мы использовали метод set_link(), чтобы определить целевую зону для только что созданной ссылки.

Чтобы создать внешнюю ссылку с помощью изображения, мы использовали метод image(). Этот метод имеет возможность передать ссылку в качестве одного из аргументов. Ссылка может быть как внутренней, так и внешней.

В качестве альтернативы для изменения стиля шрифта и добавления ссылок можно использовать метод `write_html()`. Это парсер html, который позволяет добавлять текст, изменять стиль шрифта и добавлять ссылки с помощью html.

2.8

<center>fpdf2</center>

Methods full documentation /   : `fpdf.FPDF` API doc

2.8.1 Tuto 1 -

           :

```
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

           ,   `FPDF`   FPDF

 :

     A4   :

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

                      L̀)   (`Letter`   `Legal`)
(`pt` , `cm` , `in`)

              ,   add_page

     -                             1 cm   ;   set_margins

                    ,   set_font   ,   (Document)
Helvetica bold 16   :

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

  `I`   (Italic)   , `U`   (Underlined)
         (   )   ,   (
        );   -   `Times` , `Courier` , `Symbol`   `ZapfDingbats`

   cell   cell print   (cell)   ,   ,
         ,   (   )   ,   ,
      (,   )

        ,   :

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

       (centered text)   (cell)   ,   :

```
pdf.cell(60, 10, 'Powered by FPDF.', new_x=XPos.LMARGIN, new_y=YPos.NEXT, align='C')
```

:   ln

,   output.   , output() PDF

bytearray

2.8.2 Tuto 2 -   (Header),   (Footer),   (Page Break)   (Image)

               (Header),   (Footer)   (Logo)   :

```
from fpdf import FPDF


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

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


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

Resulting PDF

               header   footer   header   footer
(automatically)   FPDF   ,   class
          override

Logo   image   -

               (Page number)   (print)   ,   (cell width)   (null value)

                                             (right margin)   ;   (center)

               page_no   ;   ,   {nb}
)

set_y   ,

                                        : the automatic page breaking.   (
          2   ),

               (Header)   (Footer)   Helvetica)   , body Times
                                                            set_auto_page_break

2.8.3 Tuto 3 -

                         Justified

```
from fpdf import FPDF


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

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

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

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

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


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

Resulting PDF

Jules Verne text

get_string_width   ,

                (set_draw_color,   )   1
(0.2   )

                (set_draw_color, set_fill_color   set_text_color   )   set_line_width   1
(0.2   )   ,   (true
)

                                                multi_cell   cell   carriage
return   ,   current cell   Text
               Justified

: (set_title) (set_author).

, , - , -

2.8.4 Tuto 4 -

,

```python
from fpdf import FPDF


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

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

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

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

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

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

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

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


pdf = PDF()
pdf.set_title("20000 Leagues Under the Seas")
pdf.set_author("Jules Verne")
```

```
pdf.print_chapter(1, "A RUNAWAY REEF", "20k_c1.txt")
pdf.print_chapter(2, "THE PROS AND CONS", "20k_c1.txt")
pdf.output("tuto4.pdf")
```

Resulting PDF

Jules Verne text

accept_page_break   set_col

accept_page_break   ,   ,
    2   (   )   set_col   ,
                                                                              accept_page_break
                                                                              ,

2.8.5 Tuto 5 -

                                            ,   -

```
import csv
from fpdf import FPDF


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

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

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


def load_data_from_csv(csv_filepath):
    headings, rows = [], []
    with open(csv_filepath, encoding="utf8") as csv_file:
        for row in csv.reader(csv_file, delimiter=","):
            if not headings:  # extracting column names from first row:
                headings = row
            else:
                rows.append(row)
    return headings, rows


col_names, data = load_data_from_csv("countries.txt")
```

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

Resulting PDF - Countries text

(Table)   (Cells)   (Collection)   ,

:   (simple framed cells,   (same
sized cells)   (left aligned cells)

:   ,
,   Cell()   ,
(L)   (R)   (Table)   Finish

:   ,   LRB   ;
,

Fill,   (Alternate)

2.8.6 Tuto 6 -

-
, (,   ,   )

```
from fpdf import FPDF, HTMLMixin


class MyFPDF(FPDF, HTMLMixin):
    pass


pdf = MyFPDF()

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

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

Resulting PDF - fpdf2-logo

write()   multi_cell()   ,   : -
-
,
,   multi_cell()

, write() ,

set_font() ,

, add_link() ,

"Link"

, - set_link()

Image , image()

, write_html() HTML ,

, 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 and orientation

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

```python
from fpdf import FPDF

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

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

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 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 pyfpdf.github.io: class Template.

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

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

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 pyfpdf.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, fields in a database). Dimensions (except font size, which always uses points) are given in user defined units (default: mm). Those are the units that can be specified when creating a `Template()` or a `FPDF()` instance.

- **name**: placeholder identification (unique text string)

- *mandatory*

- **type**:

- '**T**': Text - places one or several lines of text on the page

- '**L**': Line - draws a line from x1/y1 to x2/y2

- '**I**': Image - positions and scales an image into the bounding box

- '**B**': Box - draws a rectangle around the bounding box

- '**E**': Ellipse - draws an ellipse inside the bounding box

- '**BC**': Barcode - inserts an "Interleaved 2 of 5" type barcode

- '**C39**': Code 39 - inserts a "Code 39" type barcode

- Incompatible change: A previous implementation of this type used the non-standard element keys "x", "y", "w", and "h", which are now deprecated (but still work for the moment).

- '**W**': "Write" - uses the `FPDF.write()` method to add text to the page

- *mandatory*

- **x1, y1, x2, y2**: top-left, bottom-right coordinates, defining a bounding box in most cases

- for multiline text, this is the bounding box of just the first line, not the complete box

- for the barcodes types, the height of the barcode is `y2 - y1`, x2 is ignored.

- *mandatory* ("x2" *optional* for the barcode types)

- **font**: the name of a font type for the text types

- *optional*

- default: "helvetica"

- **size**: the size property of the element (float value)

- for text, the font size (in points!)

- for line, box, and ellipse, the line width

- for the barcode types, the width of one bar

- *optional*

- default: 10 for text, 2 for 'BC', 1.5 for 'C39'

- **bold, italic, underline**: text style properties

- in elements dict, enabled with True or equivalent value

- in csv, only int values, 0 as false, non-0 as true

- *optional*

- default: false

- **foreground, background**: text and fill colors (int value, commonly given in hex as 0xRRGGBB)

- *optional*

- default: foreground 0x000000 = black; background None/empty = transparent

- Incompatible change: Up to 2.4.5, the default background for text and box elements was solid white, with no way to make them transparent.

- **align**: text alignment, '**L**': left, '**R**': right, '**C**': center

- *optional*

- default: 'L'

- **text**: default string, can be replaced at runtime

- displayed text for 'T' and 'W'

- data to encode for barcode types

- *optional* (if missing for text/write, the element is ignored)

- default: empty

- **priority**: Z-order (int value)

- *optional*

- default: 0

- **multiline**: configure text wrapping

- in dicts, None for single line, True for multicells (multiple lines), False trims to exactly fit the space defined

- in csv, 0 for single line, >0 for multiple lines, <0 for exact fit

- *optional*

- default: single line

- **rotation**: rotate the element in degrees around the top left corner x1/y1 (float)

- *optional*

- default: 0.0 - no rotation

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

3.6 How to create a template

A template can be created in 3 ways:

- By defining everything manually in a hardcoded way as a Python dictionary

- By using a template definition in a CSV document and parsing the CSV with Template.parse_dict()

- By defining the template in a database (this applies to [Web2Py] (Web2Py.md) integration)

3.7 Example - Hardcoded

```python
from fpdf import Template

#this will define the ELEMENTS that will compose the template.
elements = [
    { 'name': 'company_logo', 'type': 'I', 'x1': 20.0, 'y1': 17.0, 'x2': 78.0, 'y2': 30.0, 'font': None, 'size': 0.0, 'bold': 0, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0, 'align': 'I', 'text': 'logo', 'priority': 2, 'multiline': 0},
    { 'name': 'company_name', 'type': 'T', 'x1': 17.0, 'y1': 32.5, 'x2': 115.0, 'y2': 37.5, 'font': 'helvetica', 'size': 12.0, 'bold': 1, 'italic': 0,
'underline': 0, 'foreground': 0, 'background': 0, 'align': 'I', 'text': '', 'priority': 2, 'multiline': 0},
    { 'name': 'multline_text', 'type': 'T', 'x1': 20, 'y1': 100, 'x2': 40, 'y2': 105, 'font': 'helvetica', 'size': 12, 'bold': 0, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0x88ff00, 'align': 'I', 'text': 'Lorem ipsum dolor sit amet, consectetur adipisici elit', 'priority': 2, 'multiline': 1}
    { 'name': 'box', 'type': 'B', 'x1': 15.0, 'y1': 15.0, 'x2': 185.0, 'y2': 260.0, 'font': 'helvetica', 'size': 0.0, 'bold': 0, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0, 'align': 'I', 'text': None, 'priority': 0, 'multiline': 0},
    { 'name': 'box_x', 'type': 'B', 'x1': 95.0, 'y1': 15.0, 'x2': 105.0, 'y2': 25.0, 'font': 'helvetica', 'size': 0.0, 'bold': 1, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0, 'align': 'I', 'text': None, 'priority': 2, 'multiline': 0},
    { 'name': 'line1', 'type': 'L', 'x1': 100.0, 'y1': 25.0, 'x2': 100.0, 'y2': 57.0, 'font': 'helvetica', 'size': 0, 'bold': 0, 'italic': 0, 'underline': 0,
'foreground': 0, 'background': 0, 'align': 'I', 'text': None, 'priority': 3, 'multiline': 0},
    { 'name': 'barcode', 'type': 'BC', 'x1': 20.0, 'y1': 246.5, 'x2': 140.0, 'y2': 254.0, 'font': 'Interleaved 2of5 NT', 'size': 0.75, 'bold': 0, 'italic': 0,
'underline': 0, 'foreground': 0, 'background': 0, 'align': 'I', 'text': '200000000001000159053338016581200810081', 'priority': 3, 'multiline': 0},
]

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

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

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

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

3.8 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;0;16777215;C;;0;0;0.0
line1;L;20.0;36.0;190.0;36.0;times;0.5;0;0;0;0;16777215;C;;0;0;0.0
name0;T;21.0;14.0;104.0;25.0;times;16.0;0;0;0;0;16777215;L;name;2;0;0.0
title0;T;21.0;26.0;104.0;30.0;times;10.0;0;0;0;0;16777215;L;title;2;0;0.0
multiline;T;21.0;50.0;28.0;54.0;times;10.5;0;0;0;0;0xffff00;L;multi line;0;1;0.0
numeric_text;T;21.0;80.0;100.0;84.0;times;10.5;0;0;0;0;;R;007;0;0;0.0
empty_fields;T;21.0;100.0;100.0;104.0
rotated;T;21.0;80.0;100.0;84.0;times;10.5;0;0;0;0;;R;ROTATED;0;0;30.0
```

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

Then you can use the file like this:

```python
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")
```

3.9 Tables

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

3.9.1 Using cells

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

```python
from fpdf import FPDF

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

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

3.9.2 Using write_html

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

```python
from fpdf import FPDF, HTMLMixin

class PDF(FPDF, HTMLMixin):
    pass

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

Note that `HTMLMixin` has some limitations, notably regarding multi-lines cells.

3.9.3 Recipes

- our 5th tutorial provides examples on how to build tables: Tuto 5 - Creating Tables
- @bvalgard wrote a custom `table()` method: YouTube video - source code

3.9.4 Repeat table header on each page

The following recipe demonstrates a solution to handle this requirement:

```python
from fpdf import FPDF

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

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

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

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

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

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

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.

method	lines	markdown support	HTML support	accepts new current position	details
`.text()`	one	no	no	fixed	Inserts a single-line text string with a precise location on the base line of the font.
`.cell()`	one	yes	no	yes	Inserts a single-line text string within the boundaries of a given box, optionally with background and border.
`.multi_cell()`	several	yes	no	yes	Inserts a multi-line text string within the boundaries of a given box, optionally with background and border.
`.write()`	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).
`.write_html()`	several	no	yes	auto	From html.py. An extension to `.write()`, with additional parsing of basic HTML tags.

4.1.1 Typographical Limitations

There are a few advanced typesetting features that fpdf doesn't currently support.

- Automatic ligatures - Some writing systems (eg. most hindic scripts such as Devaganari, Hangul) frequently combine a number of written characters into a single glyph. This would require advanced font analysis capabilities, which aren't currently implemented.
- Contextual forms - In some writing systems (eg. Arabic, Hebrew, 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.

4.1.2 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

In addition, some of the methods can optionally use markdown or HTML markup in the supplied text in order to change the font style (bold/italic/underline) of parts of the output.

4.1.3 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 are one out of the Enums `fpdf.XPos` and `fpdf.YPos` respectively, which offer the following options:

XPOS

- `XPos.LEFT` - left end of the cell
- `XPos.RIGHT` - right end of the cell
- `XPos.START` - start of actual text
- `XPos.END` - end of actual text
- `XPos.WCONT` - for write() to continue next (slightly left of END)
- `XPos.CENTER` - center of actual text
- `XPos.LMARGIN` - left page margin (start of printable area)
- `XPos.RMARGIN` - right page margin (end of printable area)

YPOS

- `YPos.TOP` - top of the first line
- `YPos.LAST` - top of the last line (same as TOP for single-line text)
- `YPos.NEXT` - top of next line (bottom of current text)
- `YPos.TMARGIN` - top page margin (start of printable area)
- `YPos.BMARGIN` - bottom page margin (end of printable area)

4.1.4 .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()

4.1.5 .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()

4.1.6 .multi_cell()

Allows printing text with 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.

Using `new_x=XPos.RIGHT, new_y=XPos.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. When `split_only == True`, returns `txt` split into lines in an array (with any markdown markup removed).

Signature and parameters for.multi_cell()

4.1.7 .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, 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()

4.1.8 .write_html()

This method can be accessed by using the class `HTMLMixin` from "html.py". It 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 consequitive 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 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, on the can use the `FPDF.unbreakable()` context-manager:

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

- `style="BI"` indicates ***bold italics***

```python
from fpdf import FPDF

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

4.4.2 .text_mode

The PDF spec defines several text modes:

TABLE 5.3   Text rendering modes

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

```python
from fpdf import FPDF, TextMode

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

with pdf.local_context(text_mode=TextMode.STROKE, line_width=2):
    pdf.cell(txt="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=TextMode.CLIP):
    pdf.cell(txt="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")
```

Hello world
CLIP text mode

More examples from `test_text_mode.py` :

- text_modes.pdf
- clip_text_modes.pdf

4.4.3 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. Using other fonts means that their variants (bold, italics) must be registered using `add_font` (with `style="B"` and `style="I"`).

```
from fpdf import FPDF

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

4.4.4 write_html

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

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

4.5 Unicode

- Unicode
- Example
- Free Font Pack and Copyright Restrictions

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

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

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

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

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

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.

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

from fpdf import FPDF

pdf = FPDF()
pdf.add_page()

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

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

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

# Add a Indic Unicode font (uses UTF-8)
# Supports: Bengali, Devanagari, Gujarati,
#           Gurmukhi (including the variants for Punjabi)
#           Kannada, Malayalam, Oriya, Tamil, Telugu, Tibetan
pdf.add_font('gargi', fname='gargi.ttf')
pdf.set_font('gargi', size=14)
pdf.write(8, u'Hindi:              ')
pdf.ln(20)

# Add a AR PL New Sung Unicode font (uses UTF-8)
```

```
# The Open Source Chinese Font (also supports other east Asian languages)
pdf.add_font('fireflysung', fname='fireflysung.ttf')
pdf.set_font('fireflysung', size=14)
pdf.write(8, u'Chinese:      \n')
pdf.write(8, u'Japanese:          \n')
pdf.ln(10)

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

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

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

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

View the result here: unicode.pdf

4.5.2 Free Font Pack and Copyright Restrictions

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

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

4.6 Emojis, Symbols & Dingbats

- Emojis, Symbols & Dingbats
- Emojis
- Symbols
- Dingbats

4.6.1 Emojis

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

```python
import fpdf

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

This code produces this PDF file: fonts_emoji_glyph.pdf

4.6.2 Symbols

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

```python
import fpdf

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

This results in:

$$\forall \, \eta \, \exists \, \varepsilon \in \Re, \eta/\varepsilon \cong \infty$$
$$\Delta \, \Phi \, \Sigma \, \Omega \, \Psi \, \alpha \, \beta \, \chi$$
$$\text{€} \, \clubsuit \, \blacklozenge \, \heartsuit \, \spadesuit \, \leftrightarrow \, \leftarrow \, \uparrow \, \rightarrow \, \downarrow \, \Leftrightarrow \, \Leftarrow \, \Rightarrow$$

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.

4.6.3 Dingbats

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

```
import fpdf

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

This results in:

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

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)
- 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 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")
```

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

5.1.3 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

5.1.4 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")
```

5.1.5 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")
```

5.1.6 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")
```

5.1.7 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")
```

5.1.8 Image compression

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

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

```python
from fpdf import FPDF

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

Beware that "flattening" images this way will fill transparent areas of your images with color (usually black).

5.1.9 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

5.1.10 Disabling transparency

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

```python
from fpdf import FPDF

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

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

5.2 Shapes

The following code snippets show examples of rendering various shapes.

5.2.1 Lines

Draw a thin plain orange line:

```python
from fpdf import FPDF

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

Draw a dashed light blue line:

```python
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 Ellipse

Draw a circle filled in grey with a pink outline:

```python
from fpdf import FPDF

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

5.2.3 Rectangle

Draw nested squares:

```python
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")
```

5.2.4 Polygon

```python
from fpdf import FPDF

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

5.2.5 Arc

```python
from fpdf import FPDF

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

5.2.6 Solid arc

```python
from fpdf import FPDF

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

5.3 Barcodes

5.3.1 Code 39

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

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

Output preview:

5.3.2 Interleaved 2 of 5

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

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

Output preview:

5.3.3 PDF-417

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

```
from pdf417 import encode, render_image

pdf = FPDF()
pdf.add_page()
img = render_image(encode("Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed non risus. Suspendisse lectus tortor, dignissim sit amet, adipiscing
nec, ultricies sed, dolor. Cras elementum ultrices diam."))
pdf.image(img, x=10, y=50)
pdf.output("pdf417.pdf")
```

Output preview:

5.3.4 QRCode

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

```python
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:

5.3.5 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.

```python
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)
```


Extend FPDF with a datamatrix() method

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

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


def datamatrix(self, text='', x=0, y=0, w=57, h=57, cellsize=5):
    "Convert text to a datamatrix barcode and put in on the page"
    encoder = DataMatrixEncoder(text)
    encoder.width = w
    encoder.height = h
    renderer = DataMatrixRenderer(encoder.matrix, encoder.regions)
    img = renderer.get_pilimage(cellsize)
    self.image(img, x, y, w, h)

FPDF.datamatrix = datamatrix
```

5.4 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).

5.4.1 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

5.4.2 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:

```python
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](#)

5.4.3 Transforms And You

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

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

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

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

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

```python
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.

5.4.4 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

5.4.5 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.

5.5 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.

5.5.1 Basic usage

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

```python
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 includes `width` and/or `height` attributes (absolute or relative), or some dimensions must be provided to `.image()` through its `w=` and/or `h=` parameters.

5.5.2 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:

```python
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:

```python
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")
```

5.5.3 Supported SVG Features

- groups
- paths

- basic shapes (rect, circle, ellipse, line, polyline, polygon)

- basic cross-references

- stroke & fill coloring and opacity

- basic stroke styling

5.5.4 Currently Unsupported Notable SVG Features

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

The biggest unsupported feature is probably:

- CSS styling of SVG elements (*cf.* issue #404)

In addition to that:

- text/tspan/textPath

- symbols

- markers

- patterns

- gradients

- embedded images or other content (including nested SVGs)

- a lot of attributes

5.6 Charts & graphs

5.6.1 Charts

Using Matplotlib

Example taken from Matplotlib artist tutorial:

```python
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)
pdf.output("matplotlib.pdf")
```

Result:
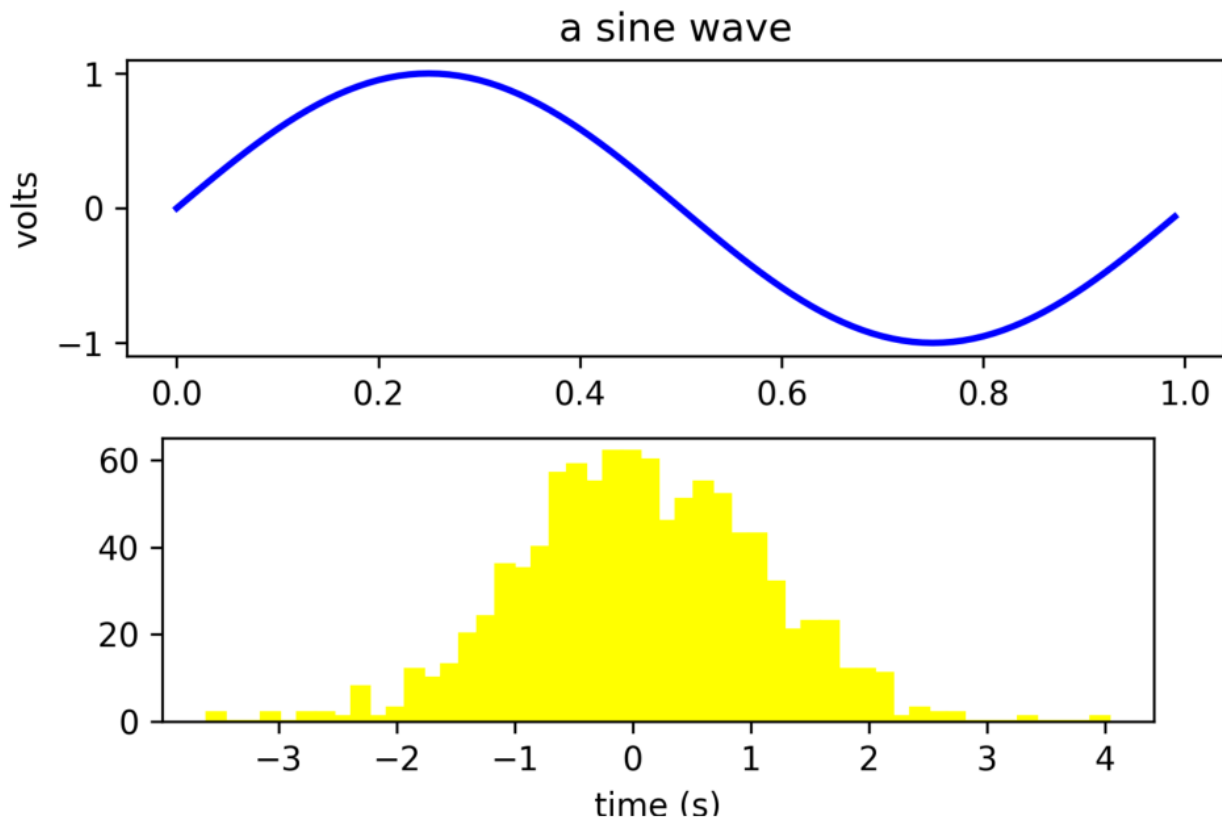
5.6.2 Mathematical formulas

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

Using Google Charts API

Official documentation: Google Charts Infographics - Mathematical Formulas.

Example:

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

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

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

Result:

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

Using Matplotlib

Matplotlib can render **LaTeX**: Text rendering With LaTeX.

Example:

```python
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)
axes = fig.gca()
axes.text(0, .5, r"$x^n + y^n = \frac{a}{b}$", fontsize=60)
axes.axis("off")

# 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=100)
pdf.output("equation-with-matplotlib.pdf")
```

Result:

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

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

6.1.2 Hyperlink with FPDF.link

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

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

```
from fpdf import FPDF

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

6.1.3 Hyperlink with write_html

An alternative method using `fpdf.HTMLMixin` :

```
from fpdf import FPDF, HTMLMixin

class PDF(FPDF, HTMLMixin):
    pass

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

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

6.1.4 Internal links

Using `FPDF.cell` :

```
from fpdf import FPDF

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

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

6.1.5 Alternative description

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

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

6.2 Metadata

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

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

- `set_title`

- `set_lang`

- `set_subject`

- `set_author`

- `set_keywords`

- `set_producer`

- `set_creator`

- `set_creation_date`

- `set_xmp_metadata`, that requires you to craft the necessary XML string

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

```python
import sys
from datetime import datetime

import pikepdf
from fpdf import FPDF_VERSION

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

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:


```python
from fpdf import FPDF

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

Method documentation: `FPDF.text_annotation`

6.3.2 Highlights

```python
from fpdf import FPDF

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

Rendering by Sumatra PDF reader:

Method documentation: `FPDF.add_highlight`

6.3.3 Named actions

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

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

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

6.3.4 Launch actions

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

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

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

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:

```python
from fpdf import FPDF

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

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

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:

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

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

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

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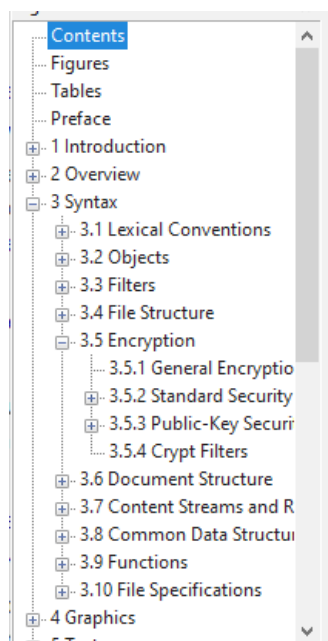
Since `fpdf2.3.3`, both features are supported through the use of the `start_section` method, that adds an entry in the internal "outline" table used to render both features.

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

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

```
from fpdf import FPDF, HTMLMixin, HTML2FPDF

class CustomHTML2FPDF(HTML2FPDF):
```

```
    def render_toc(self, pdf, outline):
        pdf.cell(txt='Table of contents:',
                new_x=XPos.LMARGIN, new_y=YPos.NEXT)
        for section in outline:
            pdf.cell(txt=f'* {section.name} (page {section.page_number})',
                    new_x=XPos.LMARGIN, new_y=YPos.NEXT)

class PDF(FPDF, HTMLMixin):
    HTML2FPDF_CLASS = CustomHTML2FPDF

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

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 test_simple_outline.pdf.

Similarly, `test_html_toc` generates test_html_toc.pdf.

7. Processing Web Content

7.1 HTML

`fpdf2` supports basic rendering from HTML.

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

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

7.1.1 write_html usage example

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

```python
from fpdf import FPDF, HTMLMixin

class PDF(FPDF, HTMLMixin):
    pass

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

7.1.2 Supported HTML features

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

- `<a>` : links (and `href` attribute)

- `<img>` : images (and `src` , `width` , `height` attributes)

- `<ol>` , `<ul>` , `<li>` : ordered, unordered and list items (can be nested)

- `<table>` : (and `border` , `width` attributes)

- `<thead>` : header (opens each page)

- `<tfoot>` : footer (closes each page)

- `<tbody>` : actual rows

- `<tr>` : rows (with `bgcolor` attribute)

- `<th>` : heading cells (with `align` , `bgcolor` , `width` attributes)

- `<td>` : cells (with `align` , `bgcolor` , `width` attributes)

Notes:

- tables should have at least a first `<th>` row with a `width` attribute.

- currently multi-line text in table cells is not supported, *cf.* issue 91. Contributions are welcome to add support for this feature! 😊

7.2 Usage in web APIs

7.2.1 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:

```python
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(txt="hello world")
    return HttpResponse(bytes(pdf.output()), content_type="application/pdf")
```

7.2.2 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`:

```python
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(txt="hello world")
    response = make_response(pdf.output())
    response.headers["Content-Type"] = "application/pdf"
    return response
```

7.2.3 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:

```python
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(txt="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",
)
```

7.2.4 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

8. Reading and Postprocessing

8.1 Existing PDFs

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

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

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

8.1.1 Adding content onto an existing PDF page

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

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

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

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

8.1.2 Adding a page to an existing PDF

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

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

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

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

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

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

8.2 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.

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

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

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

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

9. Development

9.1 Development

This page has summary information about developing the PyPDF library.

- Development
- History
- Usage
- 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
- GitHub pipeline
- Release checklist
- Documentation
- PDF spec & new features

9.1.1 History

This project, `fpdf2` is a *fork* of the `PyFPDF` project, which can be found on GitHub at reingart/pyfpdf but has been inactive since January of 2018.

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.

9.1.2 Usage

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

9.1.3 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

9.1.4 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.

9.1.5 Code auto-formatting

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

9.1.6 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.

9.1.7 Pre-commit hook

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

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

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

9.1.8 Testing

Running tests

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

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

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

Why is a test failing?

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

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

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

```
================================ FAILURES ================================
_____ test_html_simple_table _____

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

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

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

assert_pdf_equal & writing new tests

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

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

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

9.1.9 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`
3. `git commit` & `git push`
4. check that the GitHub Actions succeed, and that a new release appears on Pypi

5. perform a GitHub release, taking the description from the `CHANGELOG.md`. It will create a new `git` tag.

6. Announce the release on r/pythonnews

9.1.10 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 :
```

9.1.11 PDF spec & new features

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

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

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

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

Example output using the Tutorial first code snippet:

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

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)

```python
from fpdf import FPDF

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

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

With Reportlab:

```python
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 developped 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.