Tutorial

This tutorial will show you the use of PyMuPDF, MuPDF in Python, step by step.

Because MuPDF supports not only PDF, but also XPS, OpenXPS, CBZ, CBR, FB2 and EPUB formats, so does PyMuPDF [1]. Nevertheless, for the sake of brevity we will only talk about PDF files. At places where indeed only PDF files are supported, this will be mentioned explicitely.

Importing the Bindings

The Python bindings to MuPDF are made available by this import statement. We also show here how your version can be checked:

```
>>> import fitz
>>> print(fitz.__doc__)
PyMuPDF 1.16.0: Python bindings for the MuPDF 1.16.0 library.
Version date: 2019-07-28 07:30:14.
Built for Python 3.7 on win32 (64-bit).
```

Opening a Document

To access a supported document, it must be opened with the following statement:

```
doc = fitz.open(filename)     # or fitz.Document(filename)
```

This creates the Document object *doc*. *filename* must be a Python string specifying the name of an existing file.

It is also possible to open a document from memory data, or to create a new, empty PDF. See Document for details.

Table of Contents

A document contains many attributes and functions. Among them are meta information (like "author" or "subject"), number of total pages, outline and encryption information.

Some Document Methods and Attributes

Method / Attribute	Description
`Document.pageCount`	the number of pages (*int*)
`Document.metadata`	the metadata (*dict*)
`Document.getToC()`	get the table of contents (*list*)
`Document.loadPage()`	read a Page

Accessing Meta Data

PyMuPDF fully supports standard metadata. `Document.metadata` is a Python dictionary with the following keys. It is available for **all document types**, though not all entries may always contain data. For details of their meanings and formats consult the respective manuals, e.g. Adobe PDF References for PDF. Further information can also be found in chapter Document. The meta data fields are strings or *None* if not otherwise indicated. Also be aware that not all of them always contain meaningful data – even if they are not *None*.

Key	Value
producer	producer (producing software)
format	format: 'PDF-1.4', 'EPUB', etc.
encryption	encryption method used if any
author	author
modDate	date of last modification
keywords	keywords
title	title
creationDate	date of creation

Previous topic

Installation

Next topic

Collection of Recipes

Quick search

Go

Key	Value
creator	creating application
subject	subject

Note

Apart from these standard metadata, **PDF documents** starting from PDF version 1.4 may also contain so-called *"metadata streams"*. Information in such streams is coded in XML. PyMuPDF deliberately contains no XML components, so we do not directly support access to information contained therein. But you can extract the stream as a whole, inspect or modify it using a package like lxml and then store the result back into the PDF. If you want, you can also delete these data altogether.

Note

There are two utility scripts in the repository that import (PDF only) resp. export metadata from resp. to CSV files.

Working with Outlines

The easiest way to get all outlines (also called "bookmarks") of a document, is by loading its *table of contents*:

```
toc = doc.getToC()
```

This will return a Python list of lists *[[lvl, title, page, …], …]* which looks much like a conventional table of contents found in books.

lvl is the hierarchy level of the entry (starting from 1), *title* is the entry's title, and *page* the page number (1-based!). Other parameters describe details of the bookmark target.

Note

There are two utility scripts in the repository that import (PDF only) resp. export table of contents from resp. to CSV files.

Working with Pages

Page handling is at the core of MuPDF's functionality.

- You can render a page into a raster or vector (SVG) image, optionally zooming, rotating, shifting or shearing it.
- You can extract a page's text and images in many formats and search for text strings.
- For PDF documents many more methods are available to add text or images to pages.

First, a Page must be created. This is a method of Document:

```
page = doc.loadPage(pno)  # loads page number 'pno' of the document (0-ba
page = doc[pno]  # the short form
```

Any integer *-inf < pno < pageCount* is possible here. Negative numbers count backwards from the end, so *doc[-1]* is the last page, like with Python sequences.

Some more advanced way would be using the document as an **iterator** over its pages:

```
for page in doc:
    # do something with 'page'

# ... or read backwards
for page in reversed(doc):
    # do something with 'page'

# ... or even use 'slicing'
for page in doc.pages(start, stop, step):
    # do something with 'page'
```

Once you have your page, here is what you would typically do with it:

Inspecting the Links, Annotations or Form Fields of a Page

Links are shown as "hot areas" when a document is displayed with some viewer software. If you click while your cursor shows a hand symbol, you will usually be taken to the taget that is encoded in that hot area. Here is how to get all links:

```
# get all links on a page
links = page.getLinks()
```

links is a Python list of dictionaries. For details see `Page.getLinks()`.

You can also use an iterator which emits one link at a time:

```
for link in page.links():
    # do something with 'link'
```

If dealing with a PDF document page, there may also exist annotations (Annot) or form fields (Widget), each of which have their own iterators:

```
for annot in page.annots():
    # do something with 'annot'

for field in page.widgets():
    # do something with 'field'
```

Rendering a Page

This example creates a **raster** image of a page's content:

```
pix = page.getPixmap()
```

pix is a Pixmap object which (in this case) contains an **RGB** image of the page, ready to be used for many purposes. Method `Page.getPixmap()` offers lots of variations for controlling the image: resolution, colorspace (e.g. to produce a grayscale image or an image with a subtractive color scheme), transparency, rotation, mirroring, shifting,

shearing, etc. For example: to create an **RGBA** image (i.e. containing an alpha channel), specify *pix = page.getPixmap(alpha=True)*.

A Pixmap contains a number of methods and attributes which are referenced below. Among them are the integers *width*, *height* (each in pixels) and *stride* (number of bytes of one horizontal image line). Attribute *samples* represents a rectangular area of bytes representing the image data (a Python *bytes* object).

Note

You can also create a **vector** image of a page by using `Page.getSVGimage()`. Refer to this Wiki for details.

Saving the Page Image in a File

We can simply store the image in a PNG file:

```
pix.writeImage("page-%i.png" % page.number)
```

Displaying the Image in GUIs

We can also use it in GUI dialog managers. `Pixmap.samples` represents an area of bytes of all the pixels as a Python bytes object. Here are some examples, find more in the examples directory.

wxPython

Consult their documentation for adjustments to RGB(A) pixmaps and, potentially, specifics for your wxPython release:

```
if pix.alpha:
    bitmap = wx.Bitmap.FromBufferRGBA(pix.width, pix.height, pix.samples)
else:
    bitmap = wx.Bitmap.FromBuffer(pix.width, pix.height, pix.samples)
```

Tkinter

Please also see section 3.19 of the Pillow documentation:

```
from PIL import Image, ImageTk

# set the mode depending on alpha
mode = "RGBA" if pix.alpha else "RGB"
img = Image.frombytes(mode, [pix.width, pix.height], pix.samples)
tkimg = ImageTk.PhotoImage(img)
```

The following **avoids using Pillow**:

```
# remove alpha if present
pix1 = fitz.Pixmap(pix, 0) if pix.alpha else pix  # PPM does not support
imgdata = pix1.getImageData("ppm")  # extremely fast!
tkimg = tkinter.PhotoImage(data = imgdata)
```

If you are looking for a complete Tkinter script paging through **any supported** document, here it is! It can also zoom into pages, and it runs under Python 2 or 3. It requires the extremely handy PySimpleGUI pure Python package.

PyQt4, PyQt5, PySide

Please also see section 3.16 of the Pillow documentation:

```
from PIL import Image, ImageQt

# set the mode depending on alpha
mode = "RGBA" if pix.alpha else "RGB"
img = Image.frombytes(mode, [pix.width, pix.height], pix.samples)
qtimg = ImageQt.ImageQt(img)
```

Again, you also can get along **without using PIL** if you use the pixmap *stride* property:

```
from PyQt<x>.QtGui import QImage

# set the correct QImage format depending on alpha
fmt = QImage.Format_RGBA8888 if pix.alpha else QImage.Format_RGB888
qtimg = QImage(pix.samples, pix.width, pix.height, pix.stride, fmt)
```

Extracting Text and Images

We can also extract all text, images and other information of a page in many different forms, and levels of detail:

```
text = page.getText(opt)
```

Use one of the following strings for *opt* to obtain different formats [2]:

- *"text"*: (default) plain text with line breaks. No formatting, no text position details, no images.
- *"blocks"*: generate a list of text blocks (= paragraphs).
- *"words"*: generate a list of words (strings not containing spaces).
- *"html"*: creates a full visual version of the page including any images. This can be displayed with your internet browser.
- *"dict"* / *"json"*: same information level as HTML, but provided as a Python dictionary or resp. JSON string. See `TextPage.extractDICT()` resp. `TextPage.extractJSON()` for details of its structure.
- *"rawdict"*: a super-set of `TextPage.extractDICT()`. It additionally provides character detail information like XML. See `TextPage.extractRAWDICT()` for details of its structure.
- *"xhtml"*: text information level as the TEXT version but includes images. Can also be displayed by internet browsers.
- *"xml"*: contains no images, but full position and font information down to each single text character. Use an XML module to interpret.

To give you an idea about the output of these alternatives, we did text example extracts. See Appendix 2: Details on Text Extraction.

Searching for Text

You can find out, exactly where on a page a certain text string appears:

```
areas = page.searchFor("mupdf", hit_max = 16)
```

This delivers a list of up to 16 rectangles (see Rect), each of which surrounds one occurrence of the string "mupdf" (case insensitive). You could use this information to e.g. highlight those areas (PDF only) or create a cross reference of the document.

Please also do have a look at chapter Working together: DisplayList and TextPage and at demo programs demo.py and demo-lowlevel.py. Among other things they contain details on how the TextPage, Device and DisplayList classes can be used for a more direct control, e.g. when performance considerations suggest it.

PDF Maintenance

PDFs are the only document type that can be **modified** using PyMuPDF. Other file types are read-only.

However, you can convert **any document** (including images) to a PDF and then apply all PyMuPDF features to the conversion result. Find out more here `Document.convertToPDF()`, and also look at the demo script pdf-converter.py which can convert any supported document to PDF.

`Document.save()` always stores a PDF in its current (potentially modified) state on disk.

You normally can choose whether to save to a new file, or just append your modifications to the existing one ("incremental save"), which often is very much faster.

The following describes ways how you can manipulate PDF documents. This description is by no means complete: much more can be found in the following chapters.

Modifying, Creating, Re-arranging and Deleting Pages

There are several ways to manipulate the so-called **page tree** (a structure describing all the pages) of a PDF:

`Document.deletePage()` and `Document.deletePageRange()` delete pages.

`Document.copyPage()`, `Document.fullcopyPage()` and `Document.movePage()` copy or move a page to other locations within the same document.

`Document.select()` shrinks a PDF down to selected pages. Parameter is a sequence `[3]` of the page numbers that you want to keep. These integers must all be in range *0 <= i < pageCount*. When executed, all pages **missing** in this list will be deleted. Remaining pages will occur **in the sequence and as many times (!) as you specify them**.

So you can easily create new PDFs with

- the first or last 10 pages,
- only the odd or only the even pages (for doing double-sided printing),
- pages that **do** or **don't** contain a given text,
- reverse the page sequence, …

… whatever you can think of.

The saved new document will contain links, annotations and bookmarks that are still valid (i.a.w. either pointing to a selected page or to some external resource).

`Document.insertPage()` and `Document.newPage()` insert new pages.

Pages themselves can moreover be modified by a range of methods (e.g. page rotation, annotation and link maintenance, text and image insertion).

Joining and Splitting PDF Documents

Method `Document.insertPDF()` copies pages **between different** PDF documents. Here is a simple **joiner** example (*doc1* and *doc2* being openend PDFs):

```
# append complete doc2 to the end of doc1
doc1.insertPDF(doc2)
```

Here is a snippet that **splits** *doc1*. It creates a new document of its first and its last 10 pages:

```
doc2 = fitz.open()                  # new empty PDF
doc2.insertPDF(doc1, to_page = 9)   # first 10 pages
doc2.insertPDF(doc1, from_page = len(doc1) - 10) # last 10 pages
doc2.save("first-and-last-10.pdf")
```

More can be found in the Document chapter. Also have a look at PDFjoiner.py.

Embedding Data

PDFs can be used as containers for abitrary data (exeutables, other PDFs, text or binary files, etc.) much like ZIP archives.

PyMuPDF fully supports this feature via Document *embeddedFile** methods and attributes. For some detail read Appendix 3: Considerations on Embedded Files, consult the Wiki on embedding files, or the example scripts embedded-copy.py, embedded-export.py, embedded-import.py, and embedded-list.py.

Saving

As mentioned above, `Document.save()` will **always** save the document in its current state.

You can write changes back to the **original PDF** by specifying option *incremental=True*. This process is (usually) **extremely fast**, since changes are **appended to the original file** without completely rewriting it.

`Document.save()` options correspond to options of MuPDF's command line utility *mutool clean*, see the following table.

Save Option	mutool	Effect
garbage=1	g	garbage collect unused objects

Save Option	mutool	Effect
garbage=2	gg	in addition to 1, compact `xref` tables
garbage=3	ggg	in addition to 2, merge duplicate objects
garbage=4	gggg	in addition to 3, skip duplicate streams
clean=1	cs	clean and sanitize content streams
deflate=1	z	deflate uncompressed streams
ascii=1	a	convert binary data to ASCII format
linear=1	l	create a linearized version
expand=1	i	decompress images
expand=2	f	decompress fonts
expand=255	d	decompress all

For example, *mutool clean -ggggz file.pdf* yields excellent compression results. It corresponds to *doc.save(filename, garbage=4, deflate=1)*.

Closing

It is often desirable to "close" a document to relinquish control of the underlying file to the OS, while your program continues.

This can be achieved by the `Document.close()` method. Apart from closing the underlying file, buffer areas associated with the document will be freed.

Further Reading

Also have a look at PyMuPDF's Wiki pages. Especially those named in the sidebar under title **"Recipes"** cover over 15 topics written in "How-To" style.

This document also contains a Collection of Recipes. This chapter has close connection to the aforementioned recipes, and it will be extended with more content over time.

Footnotes

[1] PyMuPDF lets you also open several image file types just like normal documents. See section Supported Input Image Formats in chapter Pixmap for more comments.

[2] `Page.getText()` is a convenience wrapper for several methods of another PyMuPDF class, TextPage. The names of these methods correspond to the argument string passed to `Page.getText()` : *Page.getText("dict")* is equivalent to *TextPage.extractDICT()* .

[3] "Sequences" are Python objects conforming to the sequence protocol. These objects implement a method named *__getitem__()*. Best known examples are Python tuples and lists. But *array.array*, *numpy.array* and PyMuPDF's "geometry" objects (Operator Algebra for Geometry Objects) are sequences, too. Refer to Using Python Sequences as Arguments in PyMuPDF for details.