PyMuPDF Documentation

Release 1.16.7

Jorj X. McKie

Nov 08, 2019

CONTENTS

CHAPTER

ONE

INTRODUCTION

PyMuPDF is a Python binding for MuPDF[1] – "a lightweight PDF and XPS viewer".

MuPDF can access files in PDF, XPS, OpenXPS, CBZ (comic book archive), FB2 and EPUB (e-book) formats.

These are files with extensions `*.pdf`, `*.xps`, `*.oxps`, `*.cbz`, `*.fb2` or `*.epub` (so in essence, with this binding you can develop **e-book viewers in Python** ...).

PyMuPDF provides access to many important functions of MuPDF from within a Python environment, and we are continuously seeking to expand this function set.

MuPDF stands out among all similar products for its top rendering capability and unsurpassed processing speed. At the same time, its "light weight" makes it an excellent choice for platforms where resources are typically limited, like smartphones.

Check this out yourself and compare the various free PDF-viewers. In terms of speed and rendering quality SumatraPDF[2] ranges at the top (apart from MuPDF's own standalone viewer) – since it has changed its library basis to MuPDF!

While PyMuPDF has been available since several years for an earlier version of MuPDF (v1.2, called **fitz-python** then), it was until only mid May 2015, that its creator and a few co-workers decided to elevate it to support current releases of MuPDF.

PyMuPDF runs and has been tested on Mac, Linux, Windows XP SP2 and up, Python 2.7 through Python 3.7 (note that Python supports Windows XP only up to v3.4), 32bit and 64bit versions. Other platforms should work too, as long as MuPDF and Python support them.

PyMuPDF is hosted on GitHub[3]. We also are registered on PyPI[4].

For MS Windows and popular Python versions on Mac OSX and Linux we have created wheels. So installation should be convenient enough for hopefully most of our users: just issue

```
pip install --upgrade pymupdf
```

If your platform is not among those supported with a wheel, your installation consists of two separate steps:

[1] http://www.mupdf.com/
[2] http://www.sumatrapdfreader.org/
[3] https://github.com/pymupdf/PyMuPDF
[4] https://pypi.org/project/PyMuPDF/

1. Installation of MuPDF: this involves downloading the source from their website and then compiling it on your machine. Adjust `setup.py` to point to the right directories (next step), before you try generating PyMuPDF.
2. Installation of PyMuPDF: this step is normal Python procedure. Usually you will have to adapt the `setup.py` to point to correct `include` and `lib` directories of your generated MuPDF.

For installation details check out the respective chapter.

There exist several demo[5] and example[6] programs in the main repository, ranging from simple code snippets to full-featured utilities, like text extraction, PDF joiners and bookmark maintenance.

Interesting **PDF manipulation and generation** functions have been added over time, including metadata and bookmark maintenance, document restructuring, annotation / link handling and document or page creation.

1.1 Note on the Name `fitz`

The standard Python import statement for this library is `import fitz`. This has a historical reason:

The original rendering library for MuPDF was called `Libart`.

"After Artifex Software acquired the MuPDF project, the development focus shifted on writing a new modern graphics library called ''Fitz''. Fitz was originally intended as an R&D project to replace the aging Ghostscript graphics library, but has instead become the rendering engine powering MuPDF." (Quoted from Wikipedia[7]).

1.2 License

PyMuPDF is distributed under GNU GPL V3 (or later, at your choice).

MuPDF is distributed under a separate license, the **GNU AFFERO GPL V3**.

Both licenses apply, when you use PyMuPDF.

Note: Version 3 of the GNU AFFERO GPL is a lot less restrictive than its earlier versions used to be. It basically is an open source freeware license, that obliges your software to also being open source and freeware. Consult this website[8], if you want to create a commercial product with PyMuPDF.

1.3 Covered Version

This documentation covers PyMuPDF v1.16.7 features as of **2019-11-07 10:22:00**.

Note: The major and minor versions of **PyMuPDF** and **MuPDF** will always be the same. Only the third qualifier (patch level) may be different from that of MuPDF.

---

[5] https://github.com/pymupdf/PyMuPDF/tree/master/demo
[6] https://github.com/pymupdf/PyMuPDF/tree/master/examples
[7] https://en.wikipedia.org/wiki/MuPDF
[8] http://artifex.com/licensing/

CHAPTER

TWO

INSTALLATION

PyMuPDF can be installed from sources as follows or from wheels, see *Option 2: Install from Binaries*.

2.1 Option 1: Install from Sources

This is a three-step process.

2.1.1 Step 1: Download PyMuPDF

Download the sources from https://pypi.org/project/PyMuPDF/#files and decompress them.

2.1.2 Step 2: Download and Generate MuPDF

Download `mupdf-x.xx.x-source.tar.gz` from https://mupdf.com/downloads/archive and unzip / decompress it. Make sure to download the (sub-) version for which PyMuPDF has stated its compatibility.

Note: The latest MuPDF **development sources** are available on https://github.com/ArtifexSoftware/mupdf – this is **not** what you want here.

Applying any Changes and Hot Fixes to MuPDF Sources

On occasion, vital hot fixes or functional enhancements must be applied to MuPDF sources before it is generated.

Any such files are contained in the `fitz` directory of the PyMuPDF homepage[9] – their names all start with an underscore `"_"`. Currently (v1.16.x), these files and their copy destinations are the following:

- `_config.h` – PyMuPDF's configuration to control the binary file size and the inclusion of MuPDF features, see next section. This file must renamed and replace MuPDF file `/include/mupdf/fitz/config.h`. This file controls the size of the PyMuPDF binary by cutting away unneeded fonts from MuPDF.

Generate MuPDF

The MuPDF source includes generation procedures / makefiles for numerous platforms. For Windows platforms, Visual Studio solution and project definitions are provided.

PyMuPDF's homepage[10] contains additional details and hints.

[9] https://github.com/pymupdf/PyMuPDF/tree/master/fitz
[10] https://github.com/pymupdf/PyMuPDF/

2.1.3 Step 3: Build / Setup PyMuPDF

Adjust the setup.py script as necessary. E.g. make sure that:

- the include directory is correctly set in sync with your directory structure
- the object code libraries are correctly defined

Now perform a `python setup.py install`.

Note: You can also install from the sources of the Github repository. These **do not contain** the pre-generated files `fitz.py` or `fitz_wrap.c`, which instead are generated by the installation script `setup.py`. To use it, SWIG[11] must be installed on your system.

2.2 Option 2: Install from Binaries

This installation option is available for all MS Windows and the most popular 64-bit Mac OS and Linux platforms for Python versions 2.7 and 3.4 through 3.7.

Windows binaries are provided for Python 32-bit and 64-bit versions.

Mac OSX wheels are provided with the platform tag `macosx_10_6_intel`.

Linux wheels are provided with the platform tag `manylinux1_x86_64`. This makes them usable for most Linux variants like Debian, Ubuntu, etc.

Older versions can be found in the releases directory of our home page https://github.com/pymupdf/PyMuPDF/releases.

[11] https://www.swig.org/

CHAPTER

THREE

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[33]. 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.

3.1 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).
```

3.2 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.

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

[33] 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.

3.3 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*

3.4 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 Reference 1.7* 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
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[12] 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)[13] resp. export[14] metadata from resp. to CSV files.

3.5 Working with Outlines

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

[12] https://pypi.org/project/lxml/
[13] https://github.com/pymupdf/PyMuPDF/blob/master/examples/csv2meta.py
[14] https://github.com/pymupdf/PyMuPDF/blob/master/examples/meta2csv.py

```
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)[15] resp. export[16] table of contents from resp. to CSV files.

3.6 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-based)
page = doc[pno]  # the short form
```

Any integer $-\infty < pno < pageCount$ 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:

3.6.1 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:

[15] https://github.com/pymupdf/PyMuPDF/blob/master/examples/csv2toc.py
[16] https://github.com/pymupdf/PyMuPDF/blob/master/examples/toc2csv.py

```
# 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'
```

3.6.2 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[17] for details.

3.6.3 Saving the Page Image in a File

We can simply store the image in a PNG file:

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

3.6.4 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[18] directory.

[17] https://github.com/pymupdf/PyMuPDF/wiki/Vector-Image-Support
[18] https://github.com/pymupdf/PyMuPDF/tree/master/examples

3.6.4.1 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)
```

3.6.4.2 Tkinter

Please also see section 3.19 of the Pillow documentation[19]:

```
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 transparency
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![20] It can also zoom into pages, and it runs under Python 2 or 3. It requires the extremely handy PySimpleGUI[21] pure Python package.

3.6.4.3 PyQt4, PyQt5, PySide

Please also see section 3.16 of the Pillow documentation[22]:

```
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)
```

[19] https://Pillow.readthedocs.io
[20] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/doc-browser.py
[21] https://pypi.org/project/PySimpleGUI/
[22] https://Pillow.readthedocs.io

3.6.5 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[34]:

- `"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*.

3.6.6 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[23] and demo-lowlevel.py[24]. 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.

3.7 PDF Maintenance

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

[34] *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()`.

[23] https://github.com/pymupdf/PyMuPDF/blob/master/demo/demo.py

[24] https://github.com/pymupdf/PyMuPDF/blob/master/demo/demo-lowlevel.py

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[25] 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.

3.7.1 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[35] 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).

3.7.2 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)
```

[25] https://github.com/pymupdf/PyMuPDF/blob/master/demo/pdf-converter.py

[35] "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.

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[26].

3.7.3 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[27], or the example scripts embedded-copy.py[28], embedded-export.py[29], embedded-import.py[30], and embedded-list.py[31].

3.7.4 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
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)`.

[26] https://github.com/pymupdf/PyMuPDF/blob/master/examples/PDFjoiner.py
[27] https://github.com/pymupdf/PyMuPDF/wiki/Dealing-with-Embedded-Files
[28] https://github.com/pymupdf/PyMuPDF/blob/master/examples/embedded-copy.py
[29] https://github.com/pymupdf/PyMuPDF/blob/master/examples/embedded-export.py
[30] https://github.com/pymupdf/PyMuPDF/blob/master/examples/embedded-import.py
[31] https://github.com/pymupdf/PyMuPDF/blob/master/examples/embedded-list.py

3.8 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.

3.9 Further Reading

Also have a look at PyMuPDF's Wiki[32] 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.

[32] https://github.com/pymupdf/PyMuPDF/wiki

CHAPTER

FOUR

COLLECTION OF RECIPES

A collection of recipes in "How-To" format for using PyMuPDF. We aim to extend this section over time. Where appropriate we will refer to the corresponding Wiki[36] pages, but some duplication may still occur.

4.1 Images

4.1.1 How to Make Images from Document Pages

This little script will take a document filename and generate a PNG file from each of its pages.

The document can be any supported type like PDF, XPS, etc.

The script works as a command line tool which expects the filename being supplied as a parameter. The generated image files (1 per page) are stored in the directory of the script:

```
import sys, fitz                              # import the binding
fname = sys.argv[1]                           # get filename from command line
doc = fitz.open(fname)                        # open document
for page in doc:                              # iterate through the pages
    pix = page.getPixmap(alpha = False)       # render page to an image
    pix.writePNG("page-%i.png" % page.number)     # store image as a PNG
```

The script directory will now contain PNG image files named `page-0.png`, `page-1.png`, etc. Pictures have the dimension of their pages, e.g. 596 x 842 pixels for an A4 portrait sized page. They will have a resolution of 96 dpi in x and y dimension and have no transparency. You can change all that – for how to do do this, read the next sections.

4.1.2 How to Increase Image Resolution

The image of a document page is represented by a *Pixmap*, and the simplest way to create a pixmap is via method *`Page.getPixmap()`*.

This method has many options for influencing the result. The most important among them is the *Matrix*, which lets you zoom, rotate, distort or mirror the outcome.

[36] https://github.com/pymupdf/PyMuPDF/wiki

Page.getPixmap() by default will use the *Identity* matrix, which does nothing.

In the following, we apply a zoom factor of 2 to each dimension, which will generate an image with a four times better resolution for us.

```
>>> zoom_x = 2.0                       # horizontal zoom
>>> zomm_y = 2.0                       # vertical zoom
>>> mat = fitz.Matrix(zoom_x, zomm_y)  # zoom factor 2 in each dimension
>>> pix = page.getPixmap(matrix = mat) # use 'mat' instead of the identity matrix
```

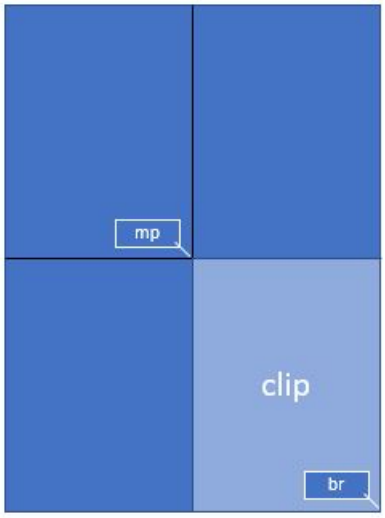
The resulting pixmap will be 4 times bigger than normal.

4.1.3 How to Create Partial Pixmaps (Clips)

You do not always need the full image of a page. This may be the case e.g. when you display the image in a GUI and would like to zoom into a part of the page.

Let's assume your GUI window has room to display a full document page, but you now want to fill this room with the bottom right quarter of your page, thus using a four times better resolution.

To achieve this, we define a rectangle equal to the area we want to appear in the GUI and call it "clip". One way of constructing rectangles in PyMuPDF is by providing two diagonally opposite corners, which is what we are doing here.




```
>>> mat = fitz.Matrix(2, 2)  # zoom factor 2 in each direction
>>> rect = page.rect  # the page rectangle
>>> mp = rect.tl + (rect.br - rect.tl) * 0.5  # its middle point
>>> clip = fitz.Rect(mp, rect.br)  # the area we want
>>> pix = page.getPixmap(matrix=mat, clip=clip)
```

In the above we construct `clip` by specifying two diagonally opposite points: the middle point `mp` of the page rectangle, and its bottom right, `rect.br`.

4.1.4 How to Create or Suppress Annotation Images

Normally, the pixmap of a page also shows the page's annotations. Occasionally, this may not be desireable.

To suppress the annotation images on a rendered page, just specify `annots=False` in *Page.getPixmap()*.

You can also render annotations separately: *Annot* objects have their own *Annot.getPixmap()* method. The resulting pixmap has the same dimensions as the annotation rectangle.

4.1.5 How to Extract Images: Non-PDF Documents

In contrast to the previous sections, this section deals with **extracting** images **contained** in documents, so they can be displayed as part of one or more pages.

If you want recreate the original image in file form or as a memory area, you have basically two options:

1. Convert your document to a PDF, and then use one of the PDF-only extraction methods. This snippet will convert a document to PDF:

```
>>> pdfbytes = doc.convertToPDF()  # this a bytes object
>>> pdf = fitz.open("pdf", pdfbytes)  # open it as a PDF document
>>> # now use 'pdf' like any PDF document
```

2. Use *Page.getText()* with the "dict" parameter. This will extract all text and images shown on the page, formatted as a Python dictionary. Every image will occur in an image block, containing meta information and the binary image data. For details of the dictionary's structure, see *TextPage*. The method works equally well for PDF files. This creates a list of all images shown on a page:

```
>>> d = page.getText("dict")
>>> blocks = d["blocks"]
>>> imgblocks = [b for b in blocks if b["type"] == 1]
```

Each item if "imgblocks" is a dictionary which looks like this:

```
{"type": 1, "bbox": (x0, y0, x1, y1), "width": w, "height": h, "ext": "png", "image": b"..."}
```

4.1.6 How to Extract Images: PDF Documents

Like any other "object" in a PDF, images are identified by a cross reference number (*xref*, an integer). If you know this number, you have two ways to access the image's data:

1. **Create** a *Pixmap* of the image with instruction `pix = fitz.Pixmap(doc, xref)`. This method is **very** fast (single digit micro-seconds). The pixmap's properties (width, height, ...) will reflect the ones of the image. In this case there is no way to tell which image format the embedded original has.
2. **Extract** the image with `img = doc.extractImage(xref)`. This is a dictionary containing the binary image data as `img["image"]`. A number of meta data are also provided – mostly the same as you would find in the pixmap of the image. The major difference is string `img["ext"]`, which specifies the image format: apart from "png", strings like "jpeg", "bmp", "tiff", etc. can also occur. Use this string as the file extension if you want to store to disk. The execution speed of this method should be compared to the combined speed of the statements `pix = fitz.Pixmap(doc,`

`xref);pix.getPNGData()`. If the embedded image is in PNG format, the speed of *Document.extractImage()* is about the same (and the binary image data are identical). Otherwise, this method is **thousands of times faster**, and the **image data is much smaller**.

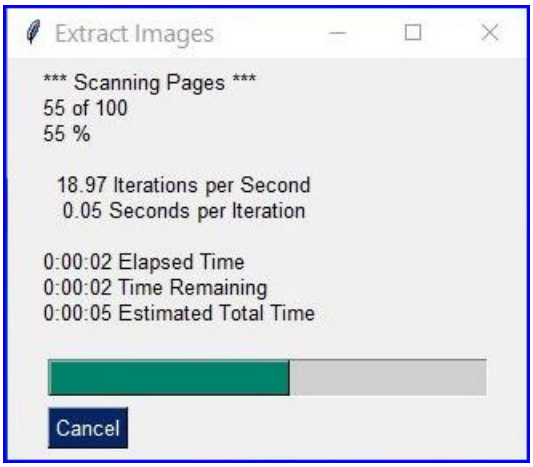
The question remains: **"How do I know those 'xref' numbers of images?"**. There are two answers to this:

a. **"Inspect the page objects:"** Loop through the items of *Page.getImageList()*. It is a list of list, and its items look like `[xref, smask, ...]`, containing the *xref* of an image. This *xref* can then be used with one of the above methods. Use this method for **valid (undamaged)** documents. Be wary however, that the same image may be referenced multiple times (by different pages), so you might want to provide a mechanism avoiding multiple extracts.

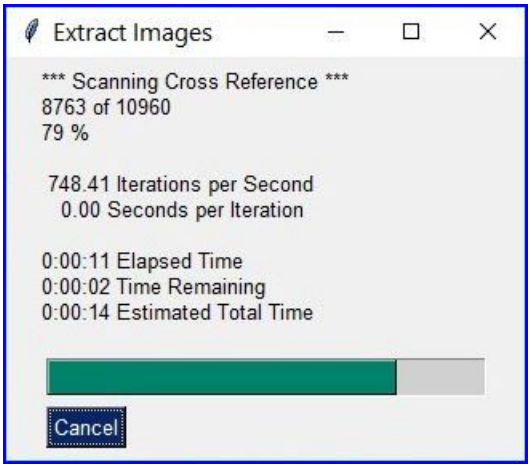
b. **"No need to know:"** Loop through the list of **all xrefs** of the document and perform a *Document.extractImage()* for each one. If the returned dictionary is empty, then continue – this *xref* is no image. Use this method if the PDF is **damaged (unusable pages)**. Note that a PDF often contains "pseudo-images" ("stencil masks") with the special purpose of defining the transparency of some other image. You may want to provide logic to exclude those from extraction. Also have a look at the next section.

For both extraction approaches, there exist ready-to-use general purpose scripts:

extract-imga.py[37] extracts images page by page:



and extract-imgb.py[38] extracts images by xref table:



[37] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/extract-imga.py
[38] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/extract-imgb.py

4.1.7 How to Handle Stencil Masks

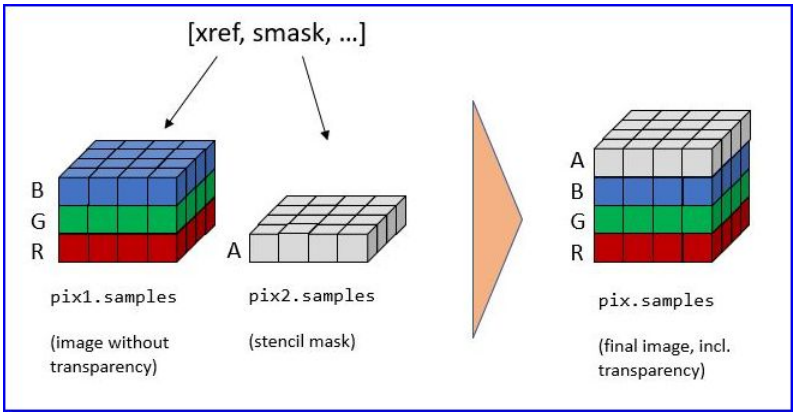
Some images in PDFs are accompanied by **stencil masks**. In their simplest form stencil masks represent alpha (transparency) bytes stored as seperate images. In order to reconstruct the original of an image, which has a stencil mask, it must be "enriched" with transparency bytes taken from its stencil mask.

Whether an image does have such a stencil mask can be recognized in one of two ways in PyMuPDF:

1. An item of *Document.getPageImageList()* has the general format `[xref, smask, ...]`, where `xref` is the image's *xref* and `smask`, if positive, is the *xref* of a stencil mask.
2. The (dictionary) results of *Document.extractImage()* have a key `"smask"`, which also contains any stencil mask's *xref* if positive.

If `smask == 0` then the image encountered via *xref* can be processed as it is.

To recover the original image using PyMuPDF, the procedure depicted as follows must be executed:




```
>>> pix1 = fitz.Pixmap(doc, xref)     # (1) pixmap of image w/o alpha
>>> pix2 = fitz.Pixmap(doc, smask)    # (2) stencil pixmap
>>> pix = fitz.Pixmap(pix1)           # (3) copy of pix1, empty alpha channel added
>>> pix.setAlpha(pix2.samples)        # (4) fill alpha channel
```

Step (1) creates a pixmap of the "netto" image. Step (2) does the same with the stencil mask. Please note that the *Pixmap.samples* attribute of `pix2` contains the alpha bytes that must be stored in the final pixmap. This is what happens in step (3) and (4).

The scripts extract-imga.py[39], and extract-imgb.py[40] above also contain this logic.

4.1.8 How to Make one PDF of all your Pictures (or Files)

We show here **three scripts** that take a list of (image and other) files and put them all in one PDF.

Method 1: Inserting Images as Pages

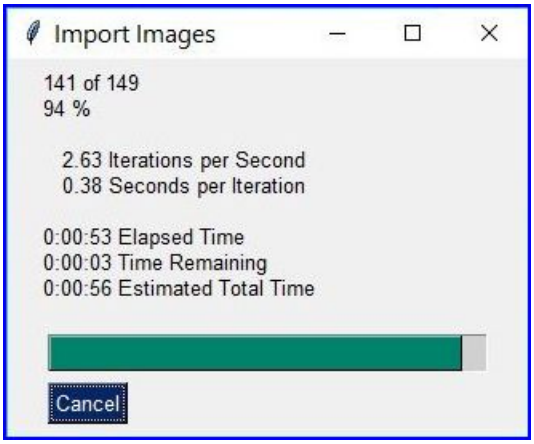
The first one converts each image to a PDF page with the same dimensions. The result will be a PDF with one page per image. It will only work for supported image file formats:

[39] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/extract-imga.py
[40] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/extract-imgb.py

```
import os, fitz
import PySimpleGUI as psg  # for showing a progress bar
doc = fitz.open()  # PDF with the pictures
imgdir = "D:/2012_10_05"  # where the pics are
imglist = os.listdir(imgdir)  # list of them
imgcount = len(imglist)  # pic count

for i, f in enumerate(imglist):
    img = fitz.open(os.path.join(imgdir, f))  # open pic as document
    rect = img[0].rect  # pic dimension
    pdfbytes = img.convertToPDF()  # make a PDF stream
    img.close()  # no longer needed
    imgPDF = fitz.open("pdf", pdfbytes)  # open stream as PDF
    page = doc.newPage(width = rect.width,  # new page with ...
                       height = rect.height)  # pic dimension
    page.showPDFpage(rect, imgPDF, 0)  # image fills the page
    psg.EasyProgressMeter("Import Images",  # show our progress
        i+1, imgcount)

doc.save("all-my-pics.pdf")
```

This will generate a PDF only marginally larger than the combined pictures' size. Some numbers on performance:

The above script needed about 1 minute on my machine for 149 pictures with a total size of 514 MB (and about the same resulting PDF size).



Look here[41] for a more complete source code: it offers a directory selection dialog and skips unsupported files and non-file entries.

Note: We might have used *Page.insertImage()* instead of *Page.showPDFpage()*, and the result would have been a similar looking file. However, depending on the image type, it may store **images uncompressed**. Therefore, the save option `deflate = True` must be used to achieve a reasonable file size, which hugely increases the runtime for large numbers of images. So this alternative **cannot be recommended** here.

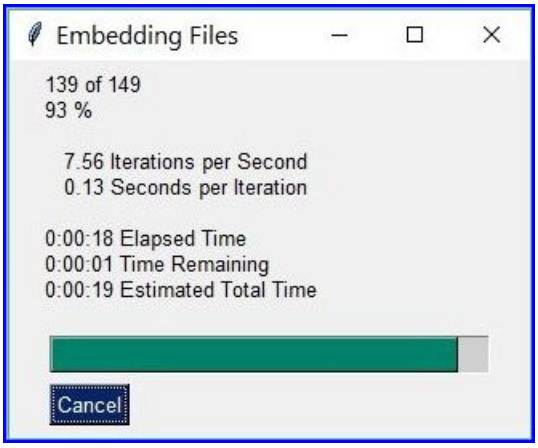
Method 2: Embedding Files

The second script **embeds** arbitrary files – not only images. The resulting PDF will have just one (empty) page, required for technical reasons. To later access the embedded files again, you would need a suitable

[41] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/all-my-pics-inserted.py

PDF viewer that can display and / or extract embedded files:

```
import os, fitz
import PySimpleGUI as psg  # for showing progress bar
doc = fitz.open()  # PDF with the pictures
imgdir = "D:/2012_10_05"  # where my files are

imglist = os.listdir(imgdir)  # list of pictures
imgcount = len(imglist)  # pic count
imglist.sort()  # nicely sort them

for i, f in enumerate(imglist):
    img = open(os.path.join(imgdir,f), "rb").read()  # make pic stream
    doc.embeddedFileAdd(img, f, filename=f,  # and embed it
                        ufilename=f, desc=f)
    psg.EasyProgressMeter("Embedding Files",  # show our progress
        i+1, imgcount)

page = doc.newPage()  # at least 1 page is needed

doc.save("all-my-pics-embedded.pdf")
```



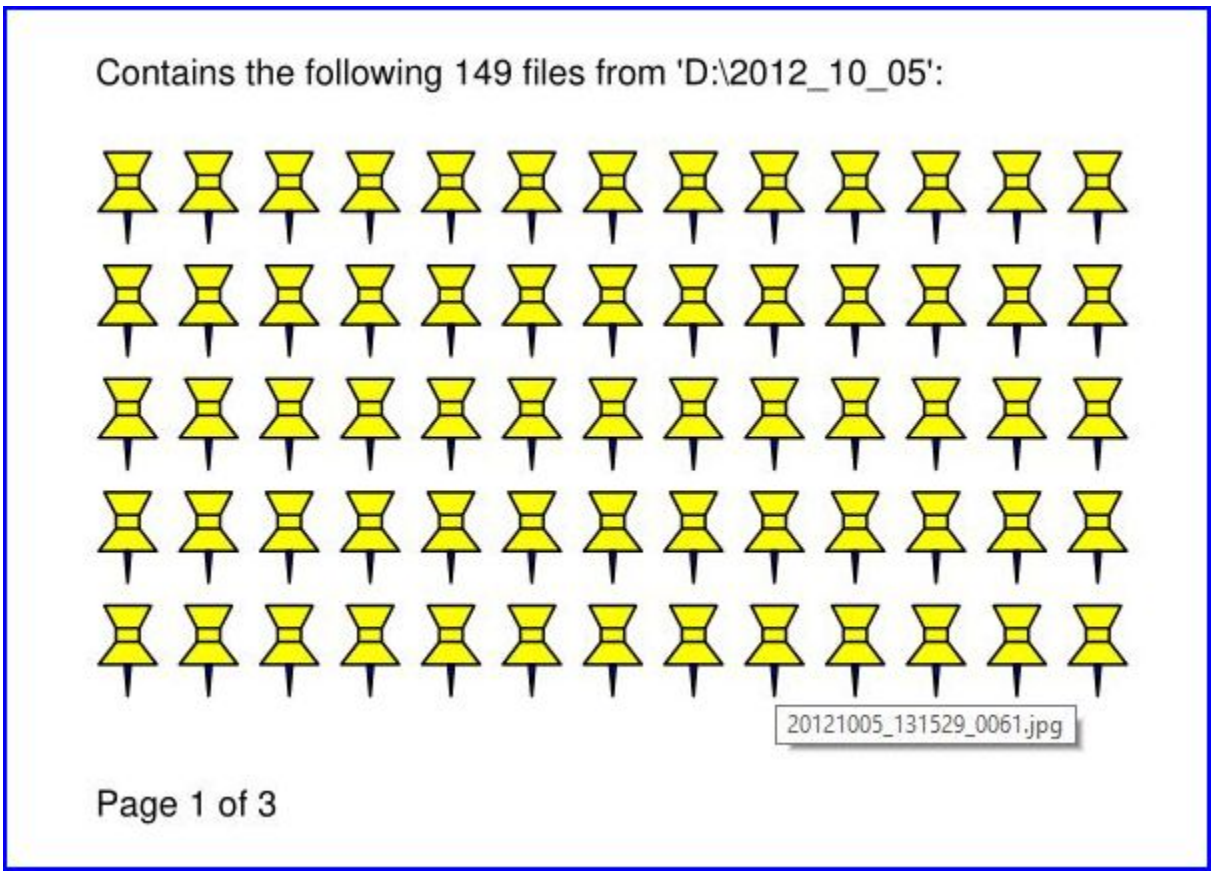
This is by far the fastest method, and it also produces the smallest possible output file size. The above pictures needed 20 seonds on my machine and yielded a PDF size of 510 MB. Look here[42] for a more complete source code: it offers a direcory selection dialog and skips non-file entries.

Method 3: Attaching Files

A third way to achieve this task is **attaching files** via page annotations see here[43] for the complete source code.

This has a similar performance as the previous script and it also produces a similar file size. It will produce PDF pages which show a 'FileAttachment' icon for each attached file.

[42] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/all-my-pics-embedded.py
[43] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/all-my-pics-attached.py



Note: Both, the **embed** and the **attach** methods can be used for **arbitrary files** – not just images.

Note: We strongly recommend using the awesome package PySimpleGUI[44] to display a progress meter for tasks that may run for an extended time span. It's pure Python, uses Tkinter (no additional GUI package) and requires just one more line of code!

4.1.9 How to Create Vector Images

The usual way to create an image from a document page is *Page.getPixmap()*. A pixmap represents a raster image, so you must decide on its quality (i.e. resolution) at creation time. It cannot be changed later.

PyMuPDF also offers a way to create a **vector image** of a page in SVG format (scalable vector graphics, defined in XML syntax). SVG images remain precise across zooming levels (of course with the exception of any raster graphic elements embedded therein).

Instruction `svg = page.getSVGimage(matrix = fitz.Identity)` delivers a UTF-8 string `svg` which can be stored with extension ".svg".

[44] https://pypi.org/project/PySimpleGUI/

4.1.10 How to Convert Images

Just as a feature among others, PyMuPDF's image conversion is easy. It may avoid using other graphics packages like PIL/Pillow in many cases.

Notwithstanding that interfacing with Pillow is almost trivial.

Input Formats	Output Formats	Description
BMP	.	Windows Bitmap
JPEG	.	Joint Photographic Experts Group
JXR	.	JPEG Extended Range
JPX	.	JPEG 2000
GIF	.	Graphics Interchange Format
TIFF	.	Tagged Image File Format
PNG	PNG	Portable Network Graphics
PNM	PNM	Portable Anymap
PGM	PGM	Portable Graymap
PBM	PBM	Portable Bitmap
PPM	PPM	Portable Pixmap
PAM	PAM	Portable Arbitrary Map
.	PSD	Adobe Photoshop Document
.	PS	Adobe Postscript

The general scheme is just the following two lines:

```
pix = fitz.Pixmap("input.xxx")  # any supported input format
pix.writeImage("output.yyy")  # any supported output format
```

Remarks

1. The **input** argument of `fitz.Pixmap(arg)` can be a file or a bytes / io.BytesIO object containing an image.
2. Instead of an output **file**, you can also create a bytes object via `pix.getImageData("yyy")` and pass this around.
3. As a matter of course, input and output formats must be compatible in terms of colorspace and transparency. The `Pixmap` class has batteries included if adjustments are needed.

Note: Convert JPEG to Photoshop:

```
pix = fitz.Pixmap("myfamily.jpg")
pix.writeImage("myfamily.psd")
```

Note: Save to JPEG using PIL/Pillow:

```
from PIL import Image
pix = fitz.Pixmap(...)
img = Image.frombytes("RGB", [pix.width, pix.height], pix.samples)
img.save("output.jpg", "JPEG")
```

Note: Convert **JPEG to Tkinter PhotoImage**. Any **RGB / no-alpha** image works exactly the same. Conversion to one of the **Portable Anymap** formats (PPM, PGM, etc.) does the trick, because they are supported by all Tkinter versions:

```
if str is bytes:                    # this is Python 2!
    import Tkinter as tk
else:                               # Python 3 or later!
    import tkinter as tk
pix = fitz.Pixmap("input.jpg")      # or any RGB / no-alpha image
tkimg = tk.PhotoImage(data=pix.getImageData("ppm"))
```

Note: Convert **PNG with alpha** to Tkinter PhotoImage. This requires **removing the alpha bytes**, before we can do the PPM conversion:

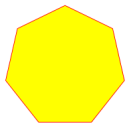
```
if str is bytes:                    # this is Python 2!
    import Tkinter as tk
else:                               # Python 3 or later!
    import tkinter as tk
pix = fitz.Pixmap("input.png")      # may have an alpha channel
if pix.alpha:                       # we have an alpha channel!
    pix = fitz.Pixmap(pix, 0)       # remove it
tkimg = tk.PhotoImage(data=pix.getImageData("ppm"))
```

4.1.11 How to Use Pixmaps: Glueing Images

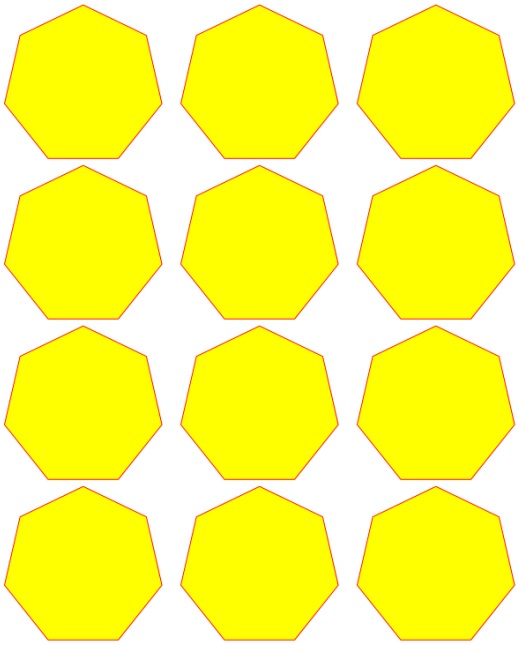
This shows how pixmaps can be used for purely graphical, non-document purposes. The script reads an image file and creates a new image which consist of 3 * 4 tiles of the original:

```
import fitz
src = fitz.Pixmap("img-7edges.png")      # create pixmap from a picture
col = 3                                  # tiles per row
lin = 4                                  # tiles per column
tar_w = src.width * col                  # width of target
tar_h = src.height * lin                 # height of target

# create target pixmap
tar_pix = fitz.Pixmap(src.colorspace, (0, 0, tar_w, tar_h), src.alpha)

# now fill target with the tiles
for i in range(col):
    src.x = src.width * i                # modify input's x coord
    for j in range(lin):
        src.y = src.height * j           # modify input's y coord
        tar_pix.copyPixmap(src, src.irect) # copy input to new loc

tar_pix.writePNG("tar.png")
```

This is the input picture:



Here is the output:



4.1.12 How to Use Pixmaps: Making a Fractal

Here is another Pixmap example that creates **Sierpinski's Carpet** – a fractal generalizing the **Cantor Set** to two dimensions. Given a square carpet, mark its 9 sub-suqares (3 times 3) and cut out the one in the center. Treat each of the remaining eight sub-squares in the same way, and continue *ad infinitum*. The end result is a set with area zero and fractal dimension 1.8928...

This script creates a approximative PNG image of it, by going down to one-pixel granularity. To increase the image precision, change the value of n (precision):

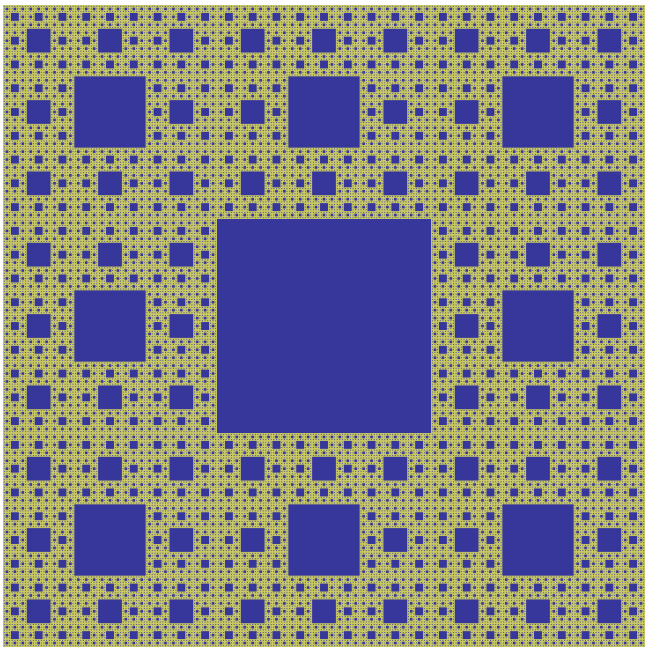
```
import fitz, time
if not list(map(int, fitz.VersionBind.split("."))) >= [1, 14, 8]:
    raise SystemExit("need PyMuPDF v1.14.8 for this script")
n = 6                               # depth (precision)
d = 3**n                            # edge length

t0 = time.perf_counter()
ir = (0, 0, d, d)                   # the pixmap rectangle

pm = fitz.Pixmap(fitz.csRGB, ir, False)
pm.setRect(pm.irect, (255,255,0)) # fill it with some background color

color = (0, 0, 255)                 # color to fill the punch holes

# alternatively, define a 'fill' pixmap for the punch holes
```

(continues on next page)

(continued from previous page)

```
# this could be anything, e.g. some photo image ...
fill = fitz.Pixmap(fitz.csRGB, ir, False) # same size as 'pm'
fill.setRect(fill.irect, (0, 255, 255))   # put some color in

def punch(x, y, step):
    """Recursively "punch a hole" in the central square of a pixmap.
    Arguments are top-left coords and the step width.
    """
    s = step // 3                 # the new step
    # iterate through the 9 sub-squares
    # the central one will be filled with the color
    for i in range(3):
        for j in range(3):
            if i != j or i != 1:  # this is not the central cube
                if s >= 3:        # recursing needed?
                    punch(x+i*s, y+j*s, s)       # recurse
            else:                 # punching alternatives are:
                pm.setRect((x+s, y+s, x+2*s, y+2*s), color)     # fill with a color
                #pm.copyPixmap(fill, (x+s, y+s, x+2*s, y+2*s))  # copy from fill
                #pm.invertIRect((x+s, y+s, x+2*s, y+2*s))       # invert colors

    return

#==============================================================================
# main program
#==============================================================================
# now start punching holes into the pixmap
punch(0, 0, d)
t1 = time.perf_counter()
pm.writeImage("sierpinski-punch.png")
t2 = time.perf_counter()
print ("%g sec to create / fill the pixmap" % round(t1-t0,3))
print ("%g sec to save the image" % round(t2-t1,3))
```

The result should look something like this:



4.1.13 How to Interface with NumPy

This shows how to create a PNG file from a numpy array (several times faster than most other methods):

```
import numpy as np
import fitz
#==============================================================================
# create a fun-colored width * height PNG with fitz and numpy
#==============================================================================
height = 150
width  = 100
bild = np.ndarray((height, width, 3), dtype=np.uint8)

for i in range(height):
    for j in range(width):
        # one pixel (some fun coloring)
        bild[i, j] = [(i+j)%256, i%256, j%256]

samples = bytearray(bild.tostring())    # get plain pixel data from numpy array
pix = fitz.Pixmap(fitz.csRGB, width, height, samples, alpha=False)
pix.writePNG("test.png")
```

4.1.14 How to Add Images to a PDF Page

There are two methods to add images to a PDF page: *Page.insertImage()* and *Page.showPDFpage()*. Both methods have things in common, but there also exist differences.

Criterion	*Page.insertImage()*	*Page.showPDFpage()*
displayable content	image file, image in memory, pixmap	PDF page
display resolution	image resolution	vectorized (except raster page content)
rotation	multiple of 90 degrees	any angle
clipping	no (full image only)	yes
keep aspect ratio	yes (default option)	yes (default option)
transparency (water marking)	depends on image	yes
location / placement	scaled to fit target rectangle	scaled to fit target rectangle
performance	automatic prevention of duplicates; MD5 calculation on every execution	automatic prevention of duplicates; faster than *Page.insertImage()*
multi-page image support	no	yes
ease of use	simple, intuitive; performance considerations apply for multiple insertions of same image	simple, intuitive; **usable for all document types** (including images!) after conversion to PDF via *Document.convertToPDF()*

Basic code pattern for *Page.insertImage()*. **Exactly one** of the parameters **filename / stream / pixmap** must be given:

```
page.insertImage(
    rect,                   # where to place the image (rect-like)
    filename=None,          # image in a file
    stream=None,            # image in memory (bytes)
    pixmap=None,            # image from pixmap
    rotate=0,               # rotate (int, multiple of 90)
    keep_proportion=True,   # keep aspect ratio
    overlay=True,           # put in foreground
)
```

Basic code pattern for *Page.showPDFpage()*. Source and target PDF must be different *Document* objects (but may be opened from the same file):

```
page.showPDFpage(
    rect,                   # where to place the image (rect-like)
    src,                    # source PDF
    pno=0,                  # page number in source PDF
    clip=None,              # only display this area (rect-like)
    rotate=0,               # rotate (float, any value)
    keep_proportion=True,   # keep aspect ratio
    overlay=True,           # put in foreground
)
```

4.2 Text

4.2.1 How to Extract all Document Text

This script will take a document filename and generate a text file from all of its text.

The document can be any supported type like PDF, XPS, etc.

The script works as a command line tool which expects the document filename supplied as a parameter. It generates one text file named "filename.txt" in the script directory. Text of pages is separated by a line "—–":

```
import sys, fitz                            # import the bindings
fname = sys.argv[1]                         # get document filename
doc = fitz.open(fname)                      # open document
out = open(fname + ".txt", "wb")            # open text output
for page in doc:                            # iterate the document pages
    text = page.getText().encode("utf8")    # get plain text (is in UTF-8)
    out.write(text)                         # write text of page
    out.write(b"\n-----\n")                 # write page delimiter
out.close()
```

The output will be plain text as it is coded in the document. No effort is made to prettify in any way. Specifally for PDF, this may mean output not in usual reading order, unexpected line breaks and so forth.

You have many options to cure this – see chapter *Appendix 2: Details on Text Extraction*. Among them are:

1. Extract text in HTML format and store it as a HTML document, so it can be viewed in any browser.
2. Extract text as a list of text blocks via `Page.getText("blocks")`. Each item of this list contains position information for its text, which can be used to establish a convenient reading order.
3. Extract a list of single words via `Page.getText("words")`. Its items are words with position information. Use it to determine text contained in a given rectangle – see next section.

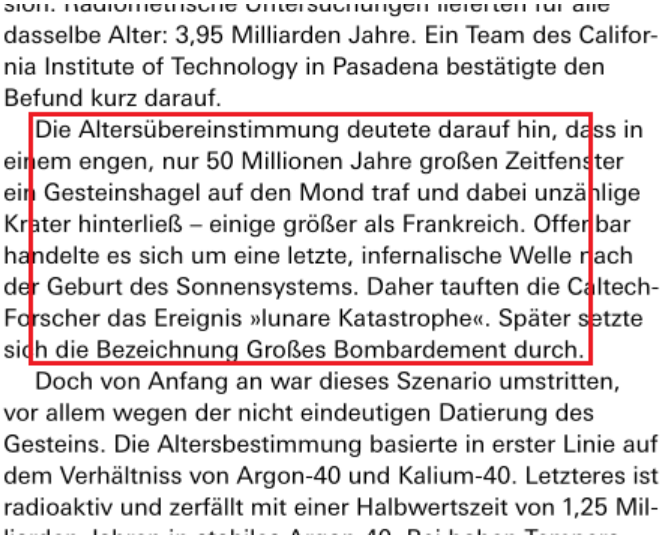
See the following two section for examples and further explanations.

4.2.2 How to Extract Text from within a Rectangle

Please refer to the script textboxtract.py[45].

It demonstrates ways to extract text contained in the following red rectangle,

[45] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/textboxtract.py


sion. Radiometrische Untersuchungen lieferten für alle
dasselbe Alter: 3,95 Milliarden Jahre. Ein Team des Califor-
nia Institute of Technology in Pasadena bestätigte den
Befund kurz darauf.
Die Altersübereinstimmung deutete darauf hin, dass in
einem engen, nur 50 Millionen Jahre großen Zeitfenster
ein Gesteinshagel auf den Mond traf und dabei unzählige
Krater hinterließ – einige größer als Frankreich. Offenbar
handelte es sich um eine letzte, infernalische Welle nach
der Geburt des Sonnensystems. Daher tauften die Caltech-
Forscher das Ereignis »lunare Katastrophe«. Später setzte
sich die Bezeichnung Großes Bombardement durch.
Doch von Anfang an war dieses Szenario umstritten,
vor allem wegen der nicht eindeutigen Datierung des
Gesteins. Die Altersbestimmung basierte in erster Linie auf
dem Verhältniss von Argon-40 und Kalium-40. Letzteres ist
radioaktiv und zerfällt mit einer Halbwertszeit von 1,25 Mil-

by using more or less restrictive conditions to find the relevant words:

```
Select the words strictly contained in rectangle
------------------------------------------------
Die Altersübereinstimmung deutete darauf hin,
engen, nur 50 Millionen Jahre großen
Gesteinshagel auf den Mond traf und dabei
hinterließ - einige größer als Frankreich.
es sich um eine letzte, infernalische Welle
Geburt des Sonnensystems. Daher tauften die
das Ereignis »lunare Katastrophe«. Später
die Bezeichnung Großes Bombardement durch.
```

Or, more forgiving, respectively:

```
Select the words intersecting the rectangle
-------------------------------------------
Die Altersübereinstimmung deutete darauf hin, dass
einem engen, nur 50 Millionen Jahre großen Zeitfenster
ein Gesteinshagel auf den Mond traf und dabei unzählige
Krater hinterließ - einige größer als Frankreich. Offenbar
handelte es sich um eine letzte, infernalische Welle nach
der Geburt des Sonnensystems. Daher tauften die Caltech-
Forscher das Ereignis »lunare Katastrophe«. Später setzte
sich die Bezeichnung Großes Bombardement durch.
```

The latter output also includes words *intersecting* the rectangle.

What if your **rectangle spans across more than one page**? Follow this recipe:

- Create a common list of all words of all pages which your rectangle intersects.
- When adding word items to this common list, increase their **y-coordinates** by the accumulated height of all previous pages.

4.2.3 How to Extract Text in Natural Reading Order

One of the common issues with PDF text extraction is, that text may not appear in any particular reading order.

Responsible for this effect is the PDF creator (software or a human). For example, page headers may have been inserted in a separate step – after the document had been produced. In such a case, the header text will appear at the end of a page text extraction (allthough it will be correctly shown by PDF viewer software). For example, the following snippet will add some header and footer lines to an existing PDF:

```
doc = fitz.open("some.pdf")
header = "Header"  # text in header
footer = "Page %i of %i"  # text in footer
for page in doc:
    page.insertText((50, 50), header)  # insert header
    page.insertText(  # insert footer 50 points above page bottom
        (50, page.rect.height - 50),
        footer % (page.number + 1, len(doc)),
    )
```

The text sequence extracted from a page modified in this way will look like this:

1. original text
2. header line
3. footer line

PyMuPDF has several means to re-establish some reading sequence or even to re-generate a layout close to the original.

As a starting point take the above mentioned script[46] and then use the full page rectangle.

On rare occasions, when the PDF creator has been "over-creative", extracted text does not even keep the correct reading sequence of **single letters**: instead of the two words "DELUXE PROPERTY" you might sometimes get an anagram, consisting of 8 words like "DEL", "XE" , "P", "OP", "RTY", "U", "R" and "E".

Such a PDF is also not searchable by all PDF viewers, but it is displayed correctly and looks harmless.

In those cases, the following function will help composing the original words of the page. The resulting list is also searchable and can be used to deliver rectangles for the found text locations:

```
from operator import itemgetter
from itertools import groupby
import fitz

def recover(words, rect):
    """ Word recovery.

    Notes:
        Method 'getTextWords()' does not try to recover words, if their single
        letters do not appear in correct lexical order. This function steps in
        here and creates a new list of recovered words.
    Args:
        words: list of words as created by 'getTextWords()'
        rect: rectangle to consider (usually the full page)
    Returns:
        List of recovered words. Same format as 'getTextWords', but left out
        block, line and word number - a list of items of the following format:
```

(continues on next page)

[46] https://github.com/pymupdf/PyMuPDF/wiki/How-to-extract-text-from-a-rectangle

(continued from previous page)

```
        [x0, y0, x1, y1, "word"]
    """
    # build my sublist of words contained in given rectangle
    mywords = [w for w in words if fitz.Rect(w[:4]) in rect]

    # sort the words by lower line, then by word start coordinate
    mywords.sort(key=itemgetter(3, 0))  # sort by y1, x0 of word rectangle

    # build word groups on same line
    grouped_lines = groupby(mywords, key=itemgetter(3))

    words_out = []  # we will return this

    # iterate through the grouped lines
    # for each line coordinate ("_"), the list of words is given
    for _, words_in_line in grouped_lines:
        for i, w in enumerate(words_in_line):
            if i == 0:  # store first word
                x0, y0, x1, y1, word = w[:5]
                continue

            r = fitz.Rect(w[:4])  # word rect

            # Compute word distance threshold as 20% of width of 1 letter.
            # So we should be safe joining text pieces into one word if they
            # have a distance shorter than that.
            threshold = r.width / len(w[4]) / 5
            if r.x0 <= x1 + threshold:  # join with previous word
                word += w[4]  # add string
                x1 = r.x1  # new end-of-word coordinate
                y0 = max(y0, r.y0)  # extend word rect upper bound
                continue

            # now have a new word, output previous one
            words_out.append([x0, y0, x1, y1, word])

            # store the new word
            x0, y0, x1, y1, word = w[:5]

        # output word waiting for completion
        words_out.append([x0, y0, x1, y1, word])

    return words_out

def search_for(text, words):
    """ Search for text in items of list of words

    Notes:
        Can be adjusted / extended in obvious ways, e.g. using regular
        expressions, or being case insensitive, or only looking for complete
        words, etc.
    Args:
        text: string to be searched for
        words: list of items in format delivered by 'getTextWords()'.
    Returns:
        List of rectangles, one for each found locations.
    """
```

(continues on next page)

(continued from previous page)

```
    rect_list = []
    for w in words:
        if text in w[4]:
            rect_list.append(fitz.Rect(w[:4]))

    return rect_list
```

4.2.4 How to Extract Tables from Documents

If you see a table in a document, you are not normally looking at something like an embedded Excel or other identifyable object. It usually is just text, formatted to appear as appropriate.

Extracting a tabular data from such a page area therefore means that you must find a way to **(1)** graphically indicate table and column borders, and **(2)** then extract text based on this information.

The wxPython GUI script wxTableExtract.py[47] strives to exactly do that. You may want to have a look at it and adjust it to your liking.

4.2.5 How to Search for and Mark Text

There is a standard search function to search for arbitrary text on a page: *Page.searchFor()*. It returns a list of *Rect* objects which surround a found occurrence. These rectangles can for example be used to automatically insert annotations which visibly mark the found text.

This method has advantages and drawbacks. Pros are

- the search string can contain blanks and wrap across lines
- upper or lower cases are treated equal
- return may also be a list of *Quad* objects to precisely locate text that is **not parallel** to either axis.

Disadvantages:

- you cannot determine the number of found items beforehand: if `hit_max` items are returned you do not know whether you have missed any.

But you have other options:

```
import sys
import fitz

def mark_word(page, text):
    """Underline each word that contains 'text'.
    """
    found = 0
    wlist = page.getTextWords()        # make the word list
    for w in wlist:                    # scan through all words on page
        if text in w[4]:               # w[4] is the word's string
            found += 1                 # count
            r = fitz.Rect(w[:4])       # make rect from word bbox
```

(continues on next page)

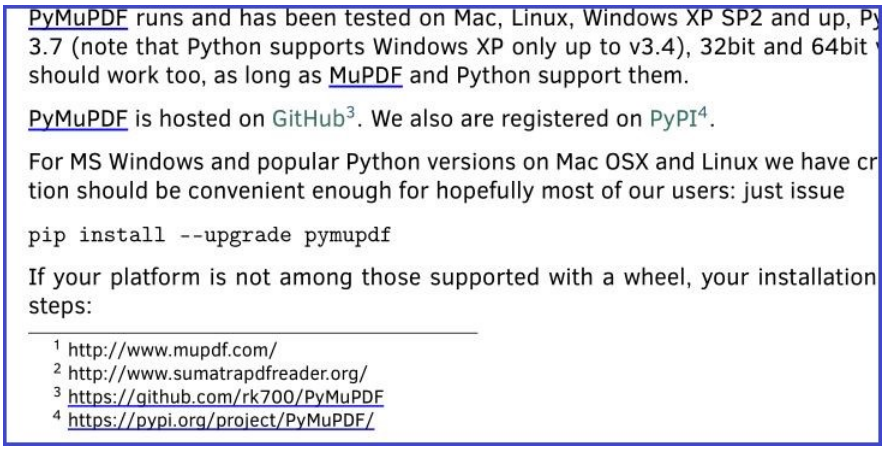
[47] https://github.com/pymupdf/PyMuPDF/blob/master/examples/wxTableExtract.py

(continued from previous page)

```
            page.addUnderlineAnnot(r)  # underline
    return found

fname = sys.argv[1]                    # filename
text = sys.argv[2]                     # search string
doc = fitz.open(fname)

print("underlining words containing '%s' in document '%s'" % (word, doc.name))

new_doc = False                        # indicator if anything found at all

for page in doc:                       # scan through the pages
    found = mark_word(page, text)      # mark the page's words
    if found:                          # if anything found ...
        new_doc = True
        print("found '%s' %i times on page %i" % (text, found, page.number + 1))

if new_doc:
    doc.save("marked-" + doc.name)
```

This script uses *Page.getTextWords()* to look for a string, handed in via cli parameter. This method separates a page's text into "words" using spaces and line breaks as delimiters. Therefore the words in this lists contain no spaces or line breaks. Further remarks:

- If found, the **complete word containing the string** is marked (underlined) – not only the search string.
- The search string may **not contain spaces** or other white space.
- As shown here, upper / lower cases are **respected**. But this can be changed by using the string method `lower()` (or even regular expressions) in function `mark_word`.
- There is **no upper limit**: all occurrences will be detected.
- You can use **anything** to mark the word: 'Underline', 'Highlight', 'StrikeThrough' or 'Square' annotations, etc.
- Here is an example snippet of a page of this manual, where "MuPDF" has been used as the search string. Note that all strings **containing "MuPDF"** have been completely underlined (not just the search string).


PyMuPDF runs and has been tested on Mac, Linux, Windows XP SP2 and up, Py
3.7 (note that Python supports Windows XP only up to v3.4), 32bit and 64bit
should work too, as long as MuPDF and Python support them.

PyMuPDF is hosted on GitHub[3]. We also are registered on PyPI[4].

For MS Windows and popular Python versions on Mac OSX and Linux we have cr
tion should be convenient enough for hopefully most of our users: just issue

```
pip install --upgrade pymupdf
```

If your platform is not among those supported with a wheel, your installation
steps:

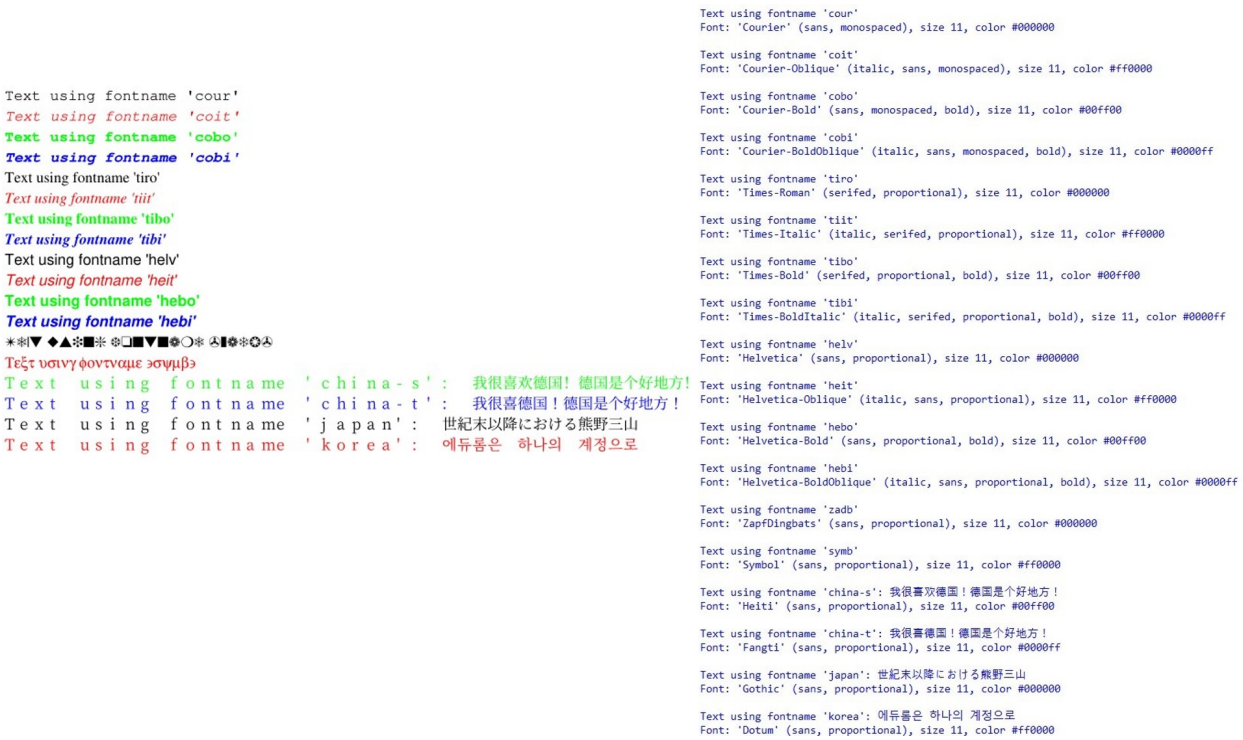
[1] http://www.mupdf.com/
[2] http://www.sumatrapdfreader.org/
[3] https://github.com/rk700/PyMuPDF
[4] https://pypi.org/project/PyMuPDF/

4.2.6 How to Analyze Font Characteristics

To analyze the characteristics of text in a PDF use this elementary script as a starting point:

```
import fitz


def flags_decomposer(flags):
    """Make font flags human readable."""
    l = []
    if flags & 2 ** 0:
        l.append("superscript")
    if flags & 2 ** 1:
        l.append("italic")
    if flags & 2 ** 2:
        l.append("serifed")
    else:
        l.append("sans")
    if flags & 2 ** 3:
        l.append("monospaced")
    else:
        l.append("proportional")
    if flags & 2 ** 4:
        l.append("bold")
    return ", ".join(l)


doc = fitz.open("text-tester.pdf")
page = doc[0]

# read page text as a dictionary, suppressing extra spaces in CJK fonts
blocks = page.getText("dict", flags=11)["blocks"]
for b in blocks:  # iterate through the text blocks
    for l in b["lines"]:  # iterate through the text lines
        for s in l["spans"]:  # iterate through the text spans
            print("")
            font_properties = "Font: '%s' (%s), size %g, color #%06x" % (
                s["font"],  # font name
                flags_decomposer(s["flags"]),  # readable font flags
                s["size"],  # font size
                s["color"],  # font color
            )
            print("Text: '%s'" % s["text"])  # simple print of text
            print(font_properties)
```

Here is the PDF page and the script output:



4.2.7 How to Insert Text

PyMuPDF provides ways to insert text on new or existing PDF pages with the following features:

- choose the font, including built-in fonts and fonts that are available as files
- choose text characteristics like bold, italic, font size, font color, etc.
- position the text in multiple ways:
  - either as simple line-oriented output starting at a certain point,
  - or fitting text in a box provided as a rectangle, in which case text alignment choices are also available,
  - choose whether text should be put in foreground (overlay existing content),
  - all text can be arbitrarily "morphed", i.e. its appearance can be changed via a *Matrix*, to achieve effects like scaling, shearing or mirroring,
  - independently from morphing and in addition to that, text can be rotated by integer multiples of 90 degrees.

All of the above is provided by three basic *Page*, resp. *Shape* methods:

- *Page.insertFont()* – install a font for the page for later reference. The result is reflected in the output of *Document.getPageFontList()*. The font can be:
  - provided as a file,
  - already present somewhere in **this or another** PDF, or
  - be a **built-in** font.

- *Page.insertText()* – write some lines of text. Internally, this uses *Shape.insertText()*.
- *Page.insertTextbox()* – fit text in a given rectangle. Here you can choose text alignment features (left, right, centered, justified) and you keep control as to whether text actually fits. Internally, this uses *Shape.insertTextbox()*.

Note: Both text insertion methods automatically install the font as necessary.

4.2.7.1 How to Write Text Lines

Output some text lines on a page:

```
import fitz
doc = fitz.open(...)                  # new or existing PDF
page = doc.newPage()                  # new or existing page via doc[n]
p = fitz.Point(50, 72)                # start point of 1st line

text = "Some text,\nspread across\nseveral lines."
# the same result is achievable by
# text = ["Some text", "spread across", "several lines."]

rc = page.insertText(p,                    # bottom-left of 1st char
                     text,                 # the text (honors '\n')
                     fontname = "helv",    # the default font
                     fontsize = 11,        # the default font size
                     rotate = 0,           # also available: 90, 180, 270
                     )
print("%i lines printed on page %i." % (rc, page.number))

doc.save("text.pdf")
```

With this method, only the **number of lines** will be controlled to not go beyond page height. Surplus lines will not be written and the number of actual lines will be returned. The calculation uses `1.2 * fontsize` as the line height and 36 points (0.5 inches) as bottom margin.

Line **width is ignored**. The surplus part of a line will simply be invisible.

However, for built-in fonts there are ways to calculate the line width beforehand - see *getTextlength()*.

Here is another example. It inserts 4 text strings using the four different rotation options, and thereby explains, how the text insertion point must be chosen to achieve the desired result:

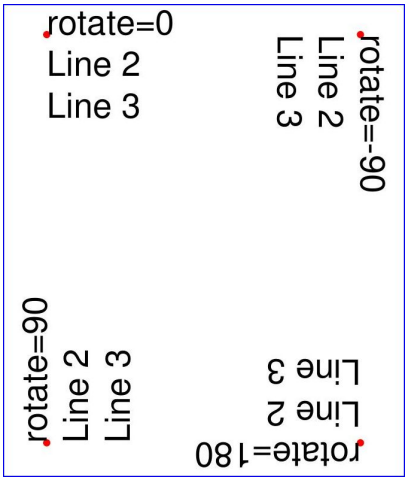
```
import fitz
doc = fitz.open()
page = doc.newPage()
# the text strings, each having 3 lines
text1 = "rotate=0\nLine 2\nLine 3"
text2 = "rotate=90\nLine 2\nLine 3"
text3 = "rotate=-90\nLine 2\nLine 3"
text4 = "rotate=180\nLine 2\nLine 3"
red = (1, 0, 0) # the color for the red dots
# the insertion points, each with a 25 pix distance from the corners
p1 = fitz.Point(25, 25)
p2 = fitz.Point(page.rect.width - 25, 25)
p3 = fitz.Point(25, page.rect.height - 25)
p4 = fitz.Point(page.rect.width - 25, page.rect.height - 25)
```

(continues on next page)

(continued from previous page)

```
# create a Shape to draw on
shape = page.newShape()

# draw the insertion points as red, filled dots
shape.drawCircle(p1,1)
shape.drawCircle(p2,1)
shape.drawCircle(p3,1)
shape.drawCircle(p4,1)
shape.finish(width=0.3, color=red, fill=red)

# insert the text strings
shape.insertText(p1, text1)
shape.insertText(p3, text2, rotate=90)
shape.insertText(p2, text3, rotate=-90)
shape.insertText(p4, text4, rotate=180)

# store our work to the page
shape.commit()
doc.save(...)
```

This is the result:



4.2.7.2 How to Fill a Text Box

This script fills 4 different rectangles with text, each time choosing a different rotation value:

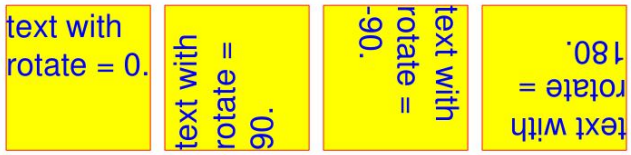
```
import fitz
doc = fitz.open(...)                          # new or existing PDF
page = doc.newPage()                          # new page, or choose doc[n]
r1 = fitz.Rect(50,100,100,150)                # a 50x50 rectangle
disp = fitz.Rect(55, 0, 55, 0)                # add this to get more rects
r2 = r1 + disp                                # 2nd rect
r3 = r1 + disp * 2                            # 3rd rect
r4 = r1 + disp * 3                            # 4th rect
t1 = "text with rotate = 0."                  # the texts we will put in
t2 = "text with rotate = 90."
t3 = "text with rotate = -90."
```

(continues on next page)

(continued from previous page)

```
t4 = "text with rotate = 180."
red  = (1,0,0)                                  # some colors
gold = (1,1,0)
blue = (0,0,1)
"""We use a Shape object (something like a canvas) to output the text and
the rectangles surounding it for demonstration.
"""
shape = page.newShape()                         # create Shape
shape.drawRect(r1)                              # draw rectangles
shape.drawRect(r2)                              # giving them
shape.drawRect(r3)                              # a yellow background
shape.drawRect(r4)                              # and a red border
shape.finish(width = 0.3, color = red, fill = gold)
# Now insert text in the rectangles. Font "Helvetica" will be used
# by default. A return code rc < 0 indicates insufficient space (not checked here).
rc = shape.insertTextbox(r1, t1, color = blue)
rc = shape.insertTextbox(r2, t2, color = blue, rotate = 90)
rc = shape.insertTextbox(r3, t3, color = blue, rotate = -90)
rc = shape.insertTextbox(r4, t4, color = blue, rotate = 180)
shape.commit()                                  # write all stuff to page /Contents
doc.save("...")
```

Several default values were used above: font “Helvetica”, font size 11 and text alignment “left”. The result will look like this:



4.2.7.3 How to Use Non-Standard Encoding

Since v1.14, MuPDF allows Greek and Russian encoding variants for the *Base14_Fonts*. In PyMuPDF this is supported via an additional `encoding` argument. Effectively, this is relevant for Helvetica, Times-Roman and Courier (and their bold / italic forms) and characters outside the ASCII code range only. Elsewhere, the argument is ignored. Here is how to request Russian encoding with the standard font Helvetica:

```
page.insertText(point, russian_text, encoding=fitz.TEXT_ENCODING_CYRILLIC)
```

The valid encoding values are TEXT_ENCODING_LATIN (0), TEXT_ENCODING_GREEK (1), and TEXT_ENCODING_CYRILLIC (2, Russian) with Latin being the default. Encoding can be specified by all relevant font and text insertion methods.

By the above statement, the fontname `helv` is automatically connected to the Russian font variant of Helvetica. Any subsequent text insertion with **this fontname** will use the Russian Helvetica encoding.

If you change the fontname just slightly, you can also achieve an **encoding “mixture”** for the **same base font** on the same page:

```
import fitz
doc=fitz.open()
page = doc.newPage()
```

(continues on next page)

(continued from previous page)

```
shape = page.newShape()
t="Sômé tèxt wìth nöñ-Lâtîn characterß."
shape.insertText((50,70), t, fontname="helv", encoding=fitz.TEXT_ENCODING_LATIN)
shape.insertText((50,90), t, fontname="HElv", encoding=fitz.TEXT_ENCODING_GREEK)
shape.insertText((50,110), t, fontname="HELV", encoding=fitz.TEXT_ENCODING_CYRILLIC)
shape.commit()
doc.save("t.pdf")
```

The result:

Sômé tèxt wìth nöñ-Lâtîn characterß.

Sтmı tθxt wμth nφρ-Lβtξn characteri̇.

STmИ tXxt wЛth nЖЯ-LБtHn characteъ.

The snippet above indeed leads to three different copies of the Helvetica font in the PDF. Each copy is uniquely idetified (and referenceable) by using the correct upper-lower case spelling of the reserved word "helv":

```
for f in doc.getPageFontList(0): print(f)

[6, 'n/a', 'Type1', 'Helvetica', 'helv', 'WinAnsiEncoding']
[7, 'n/a', 'Type1', 'Helvetica', 'HElv', 'WinAnsiEncoding']
[8, 'n/a', 'Type1', 'Helvetica', 'HELV', 'WinAnsiEncoding']
```

4.3 Annotations

In v1.14.0, annotation handling has been considerably extended:

- New annotation type support for 'Ink', 'Rubber Stamp' and 'Squiggly' annotations. Ink annots simulate handwritings by combining one or more lists of interconnected points. Stamps are intended to visually inform about a document's status or intended usage (like "draft", "confidential", etc.). 'Squiggly' is a text marker annot, which underlines selected text with a zigzagged line.
- **Extended 'FreeText' support:**
    1. all characters from the `Latin` character set are now available,
    2. colors of text, rectangle background and rectangle border can be independently set
    3. text in rectangle can be rotated by either +90 or -90 degrees
    4. text is automatically wrapped (made multi-line) in available rectangle
    5. all Base-14 fonts are now available (*normal* variants only, i.e. no bold, no italic).
- MuPDF now supports line end icons for 'Line' annots (only). PyMuPDF supported that in v1.13.x already – and for (almost) the full range of applicable types. So we adjusted the appearance of 'Polygon' and 'PolyLine' annots to closely resemble the one of MuPDF for 'Line'.
- MuPDF now provides its own annotation icons where relevant. PyMuPDF switched to using them (for 'FileAttachment' and 'Text' ["sticky note"] so far).

- MuPDF now also supports 'Caret', 'Movie', 'Sound' and 'Signature' annotations, which we may include in PyMuPDF at some later time.

4.3.1 How to Add and Modify Annotations

In PyMuPDF, new annotations are added via *Page* methods. To keep code duplication effort small, we only offer a minimal set of options here. For example, to add a 'Circle' annotation, only the containing rectangle can be specified. The result is a circle (or ellipsis) with white interior, black border and a line width of 1, exactly fitting into the rectangle. To adjust the annot's appearance, *Annot* methods must then be used. After having made all required changes, the annot's *Annot.update()* methods must be invoked to finalize all your changes.

As an overview for these capabilities, look at the following script that fills a PDF page with most of the available annotations. Look in the next sections for more special situations:

```
# -*- coding: utf-8 -*-
from __future__ import print_function
import sys
print("Python", sys.version, "on", sys.platform, "\n")
import fitz
print(fitz.__doc__, "\n")

text = "text in line\ntext in line\ntext in line\ntext in line"
red    = (1, 0, 0)
blue   = (0, 0, 1)
gold   = (1, 1, 0)
colors = {"stroke": blue, "fill": gold}
colors2 = {"fill": blue, "stroke": gold}
border = {"width": 0.3, "dashes": [2]}
displ = fitz.Rect(0, 50, 0, 50)
r = fitz.Rect(50, 100, 220, 135)
t1 = u"têxt üsès Lätiñ charß,\nEUR: €, mu: µ, super scripts: ²³!"

def print_descr(rect, annot):
    """Print a short description to the right of an annot rect."""
    annot.parent.insertText(rect.br + (10, 0),
                    "'%s' annotation" % annot.type[1], color = red)

def rect_from_quad(q):
    """Create a rect envelopping a quad (= rotated rect)."""
    return fitz.Rect(q[0], q[1]) | q[2] | q[3]

doc = fitz.open()
page = doc.newPage()
annot = page.addFreetextAnnot(r, t1, rotate = 90)
annot.setBorder(border)
annot.update(fontsize = 10, border_color=red, fill_color=gold, text_color=blue)

print_descr(annot.rect, annot)
r = annot.rect + displ
print("added 'FreeText'")

annot = page.addTextAnnot(r.tl, t1)
annot.setColors(colors2)
annot.update()
print_descr(annot.rect, annot)
```

(continues on next page)

(continued from previous page)

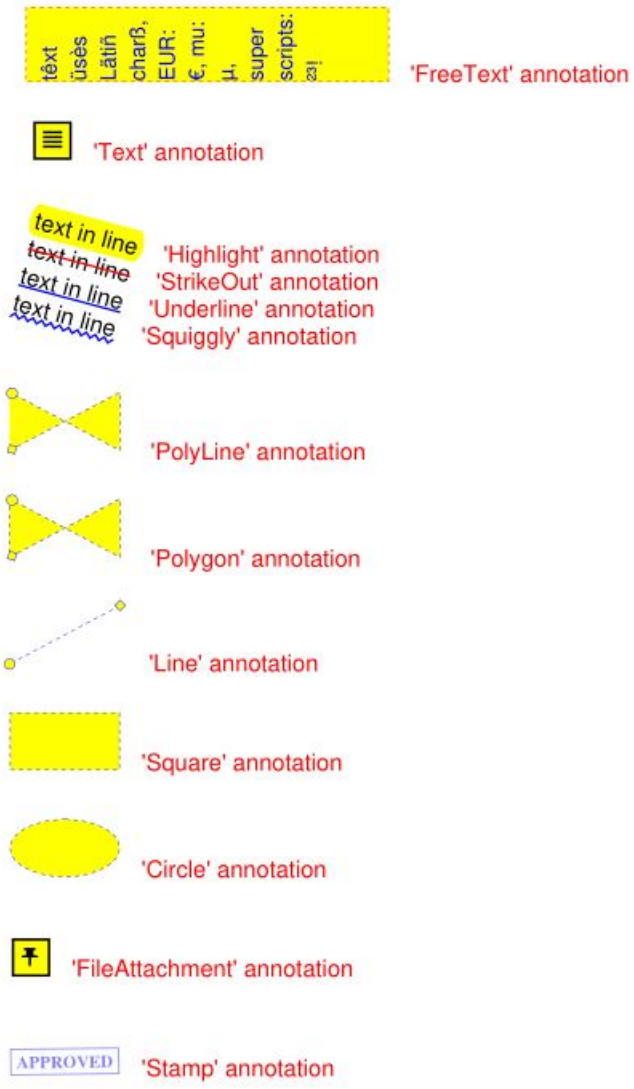
```
print("added 'Sticky Note'")

pos = annot.rect.tl + displ.tl

# first insert 4 text lines, rotated clockwise by 15 degrees
page.insertText(pos, text, fontsize=11, morph = (pos, fitz.Matrix(-15)))
# now search text to get the quads
rl = page.searchFor("text in line", quads = True)
r0 = rl[0]
r1 = rl[1]
r2 = rl[2]
r3 = rl[3]
annot = page.addHighlightAnnot(r0)
# need to convert quad to rect for descriptive text ...
print_descr(rect_from_quad(r0), annot)
print("added 'HighLight'")

annot = page.addStrikeoutAnnot(r1)
print_descr(rect_from_quad(r1), annot)
print("added 'StrikeOut'")

annot = page.addUnderlineAnnot(r2)
print_descr(rect_from_quad(r2), annot)
print("added 'Underline'")

annot = page.addSquigglyAnnot(r3)
print_descr(rect_from_quad(r3), annot)
print("added 'Squiggly'")

r = rect_from_quad(r3) + displ
annot = page.addPolylineAnnot([r.bl, r.tr, r.br, r.tl])
annot.setBorder(border)
annot.setColors(colors)
annot.setLineEnds(fitz.ANNOT_LE_Diamond, fitz.ANNOT_LE_Circle)
annot.update()
print_descr(annot.rect, annot)
print("added 'PolyLine'")

r+= displ
annot = page.addPolygonAnnot([r.bl, r.tr, r.br, r.tl])
annot.setBorder(border)
annot.setColors(colors)
annot.setLineEnds(fitz.ANNOT_LE_Diamond, fitz.ANNOT_LE_Circle)
annot.update()
print_descr(annot.rect, annot)
print("added 'Polygon'")

r+= displ
annot = page.addLineAnnot(r.tr, r.bl)
annot.setBorder(border)
annot.setColors(colors)
annot.setLineEnds(fitz.ANNOT_LE_Diamond, fitz.ANNOT_LE_Circle)
annot.update()
print_descr(annot.rect, annot)
print("added 'Line'")

r+= displ
```

(continues on next page)

(continued from previous page)

```
annot = page.addRectAnnot(r)
annot.setBorder(border)
annot.setColors(colors)
annot.update()
print_descr(annot.rect, annot)
print("added 'Square'")

r+= displ
annot = page.addCircleAnnot(r)
annot.setBorder(border)
annot.setColors(colors)
annot.update()
print_descr(annot.rect, annot)
print("added 'Circle'")

r+= displ
annot = page.addFileAnnot(r.tl, b"just anything for testing", "testdata.txt")
annot.setColors(colors2)
annot.update()
print_descr(annot.rect, annot)
print("added 'FileAttachment'")

r+= displ
annot = page.addStampAnnot(r, stamp = 0)
annot.setColors(colors)
annot.setOpacity(0.5)
annot.update()
print_descr(annot.rect, annot)
print("added 'Stamp'")

doc.save("new-annots.pdf", expand=255)
```

This script should lead to the following output:



4.3.2 How to Mark Text

This script searches for text and marks it:

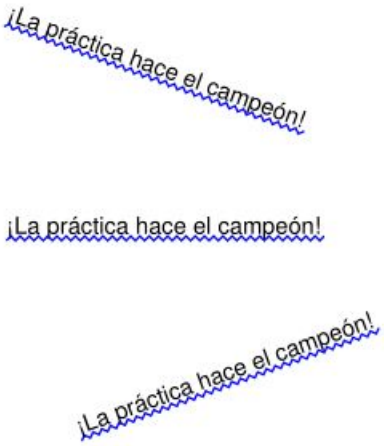
```
# -*- coding: utf-8 -*-
import fitz

# the document to annotate
doc = fitz.open("tilted-text.pdf")

# the text to be marked
t = "¡La práctica hace el campeón!"

# work with first page only
page = doc[0]
```

(continues on next page)

(continued from previous page)

```
# get list of text locations
# we use "quads", not rectangles because text may be tilted!
rl = page.searchFor(t, quads = True)

# mark all found quads with one annotation
page.addSquigglyAnnot(rl)

# save to a new PDF
doc.save("a-squiggly.pdf")
```

The result looks like this:



4.3.3 How to Use FreeText

This script shows a couple of possibilities for 'FreeText' annotations:

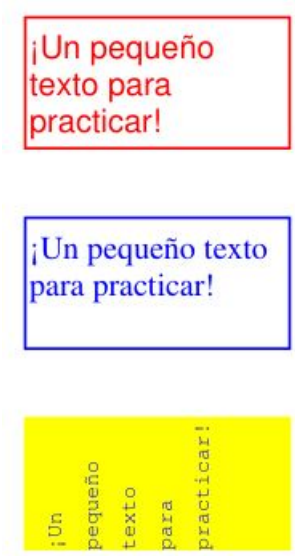
```
# -*- coding: utf-8 -*-
import fitz

# some colors
blue  = (0,0,1)
green = (0,1,0)
red   = (1,0,0)
gold  = (1,1,0)

# a new PDF with 1 page
doc = fitz.open()
page = doc.newPage()

# 3 rectangles, same size, abvove each other
r1 = fitz.Rect(100,100,200,150)
r2 = r1 + (0,75,0,75)
r3 = r2 + (0,75,0,75)
```

(continues on next page)

(continued from previous page)

```
# the text, Latin alphabet
t = "¡Un pequeño texto para practicar!"

# add 3 annots, modify the last one somewhat
a1 = page.addFreetextAnnot(r1, t, color=red)
a2 = page.addFreetextAnnot(r2, t, fontname="Ti", color=blue)
a3 = page.addFreetextAnnot(r3, t, fontname="Co", color=blue, rotate=90)
a3.setBorder({"width":0.0})
a3.update(fontsize=8, fill_color=gold)

# save the PDF
doc.save("a-freetext.pdf")
```

The result looks like this:



4.3.4 How to Use Ink Annotations

Ink annotations are used to contain freehand scribblings. A typical example maybe an image of your signature consisting of first name and last name. Technically an ink annotation is implemented as a **list of lists of points**. Each point list is regarded as a continuous line connecting the points. Different point lists represent indepndent line segments of the annotation.

The following script creates an ink annotation with two mathematical curves (sine and cosine function graphs) as line segments:

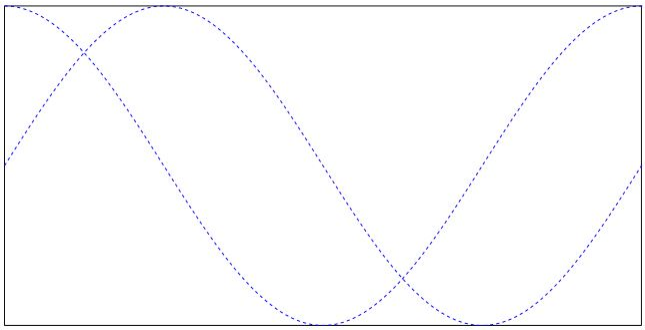
```
import math
import fitz

#----------------------------------------------------------------------------
# preliminary stuff: create function value lists for sine and cosine
#----------------------------------------------------------------------------
w360 = math.pi * 2                          # go through full circle
```

(continues on next page)

(continued from previous page)

```
deg = w360 / 360                            # 1 degree as radiants
rect = fitz.Rect(100,200, 300, 300)         # use this rectangle
first_x = rect.x0                           # x starts from left
first_y = rect.y0 + rect.height / 2.        # rect middle means y = 0
x_step = rect.width / 360                   # rect width means 360 degrees
y_scale = rect.height / 2.                  # rect height means 2
sin_points = []                             # sine values go here
cos_points = []                             # cosine values go here
for x in range(362):                        # now fill in the values
    x_coord = x * x_step + first_x          # current x coordinate
    y = -math.sin(x * deg)                  # sine
    p = (x_coord, y * y_scale + first_y)    # corresponding point
    sin_points.append(p)                    # append
    y = -math.cos(x * deg)                  # cosine
    p = (x_coord, y * y_scale + first_y)    # corresponding point
    cos_points.append(p)                    # append

#-----------------------------------------------------------------------------
# create the document with one page
#-----------------------------------------------------------------------------
doc = fitz.open()                           # make new PDF
page = doc.newPage()                        # give it a page

#-----------------------------------------------------------------------------
# add the Ink annotation, consisting of 2 curve segments
#-----------------------------------------------------------------------------
annot = page.addInkAnnot((sin_points, cos_points))
# let it look a little nicer
annot.setBorder({"width":0.3, "dashes":[1]})# line thickness, some dashing
annot.setColors({"stroke":(0,0,1)})         # make the lines blue
annot.update()                              # update the appearance

# expendable, only shows that we actually hit the rectangle
page.drawRect(rect, width = 0.3)            # only to demonstrate we did OK

doc.save("a-inktest.pdf")
```

This is the result:



4.4 Drawing and Graphics

PDF files support elementary drawing operations as part of their syntax. This includes basic geometrical objects like lines, curves, circles, rectangles including specifying colors.

The syntax for such operations is defined in "A Operator Summary" on page 985 of the *Adobe PDF Reference 1.7*. Specifying these operators for a PDF page happens in its *contents* objects.

PyMuPDF implements a large part of the available features via its *Shape* class, which is comparable to notions like "canvas" in other packages (e.g. reportlab[48]).

A shape is always created as a **child of a page**, usually with an instruction like `shape = page.newShape()`. The class defines numerous methods that perform drawing operations on the page's area. For example, `last_point = shape.drawRect(rect)` draws a rectangle along the borders of a suitably defined `rect = fitz.Rect(...)`.

The returned `last_point` **always** is the *Point* where drawing operation ended ("last point"). Every such elementary drawing requires a subsequent *Shape.finish()* to "close" it, but there may be multiple drawings which have one common `finish()` method.

In fact, *Shape.finish()* *defines* a group of preceding draw operations to form one – potentially rather complex – graphics object. PyMuPDF provides several predefined graphics in shapes_and_symbols.py[49] which demonstrate how this works.

If you import this script, you can also directly use its graphics as in the following exmple:

```
# -*- coding: utf-8 -*-
"""
Created on Sun Dec  9 08:34:06 2018

@author: Jorj
@license: GNU GPL 3.0+

Create a list of available symbols defined in shapes_and_symbols.py

This also demonstrates an example usage: how these symbols could be used
as bullet-point symbols in some text.

"""

import fitz
import shapes_and_symbols as sas

# list of available symbol functions and their descriptions
tlist = [
         (sas.arrow, "arrow (easy)"),
         (sas.caro, "caro (easy)"),
         (sas.clover, "clover (easy)"),
         (sas.diamond, "diamond (easy)"),
         (sas.dontenter, "do not enter (medium)"),
         (sas.frowney, "frowney (medium)"),
         (sas.hand, "hand (complex)"),
         (sas.heart, "heart (easy)"),
         (sas.pencil, "pencil (very complex)"),
         (sas.smiley, "smiley (easy)"),
         ]
```

(continues on next page)

[48] https://pypi.org/project/reportlab/
[49] https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/shapes_and_symbols.py

(continued from previous page)

```
r = fitz.Rect(50, 50, 100, 100)         # first rect to contain a symbol
d = fitz.Rect(0, r.height + 10, 0, r.height + 10) # displacement to next ret
p = (15, -r.height * 0.2)               # starting point of explanation text
rlist = [r]                             # rectangle list

for i in range(1, len(tlist)):          # fill in all the rectangles
    rlist.append(rlist[i-1] + d)

doc = fitz.open()                       # create empty PDF
page = doc.newPage()                    # create an empty page
shape = page.newShape()                   # start a Shape (canvas)

for i, r in enumerate(rlist):
    tlist[i][0](shape, rlist[i])          # execute symbol creation
    shape.insertText(rlist[i].br + p,     # insert description text
                     tlist[i][1], fontsize=r.height/1.2)

# store everything to the page's /Contents object
shape.commit()

import os
scriptdir = os.path.dirname(__file__)
doc.save(os.path.join(scriptdir, "symbol-list.pdf"))  # save the PDF
```

This is the script's outcome:

4.5 Multiprocessing

MuPDF has no integrated support for threading - they call themselves "threading-agnostic". While there do exist tricky possibilities to still use threading with MuPDF, the baseline consequence for **PyMuPDF** is:

No Python threading support.

Using PyMuPDF in a Python threading environment will lead to blocking effects for the main thread.

However, there exists the option to use Python's `multiprocessing` module in a variety of ways.

If you are looking to speed up page-oriented processing for a large document, use this script as a starting point. It should be at least twice as fast as the corresponding sequential processing.

```
"""
Demonstrate the use of multiprocessing with PyMuPDF.

Depending on the  number of CPUs, the document is divided in page ranges.
Each range is then worked on by one process.
The type of work would typically be text extraction or page rendering. Each
process must know where to put its results, because this processing pattern
does not include inter-process communication or data sharing.

Compared to sequential processing, speed improvements in range of 100% (ie.
twice as fast) or better can be expected.
"""
from __future__ import print_function, division
import sys
import os
import time
from multiprocessing import Pool, cpu_count
import fitz

# choose a version specific timer function (bytes == str in Python 2)
mytime = time.clock if str is bytes else time.perf_counter


def render_page(vector):
    """ Render a page range of a document.

    Notes:
        The PyMuPDF document cannot be part of the argument, because that
        cannot be pickled. So we are being passed in just its filename.
        This is no performance issue, because we are a separate process and
        need to open the document anyway.
        Any page-specific function can be processed here - rendering is just
        an example - text extraction might be another.
        The work must however be self-contained: no inter-process communication
        or synchronization is possible with this design.
        Care must also be taken with which parameters are contained in the
        argument, because it will be passed in via pickling by the Pool class.
        So any large objects will increase the overall duration.
    Args:
        vector: a list containing required parameters.
    """
    # recreate the arguments
    idx = vector[0]  # this is the segment number we have to process
    cpu = vector[1]  # number of CPUs
```

(continues on next page)

(continued from previous page)

```
    filename = vector[2]  # document filename
    mat = vector[3]  # the matrix for rendering
    doc = fitz.open(filename)  # open the document
    num_pages = len(doc)  # get number of pages

    # pages per segment: make sure that cpu * seg_size >= num_pages!
    seg_size = int(num_pages / cpu + 1)
    seg_from = idx * seg_size  # our first page number
    seg_to = min(seg_from + seg_size, num_pages)  # last page number

    for i in range(seg_from, seg_to):  # work through our page segment
        page = doc[i]
        # page.getText("rawdict")  # use any page-related type of work here, eg
        pix = page.getPixmap(alpha=False, matrix=mat)
        # store away the result somewhere ...
        # pix.writePNG("p-%i.png" % i)
    print("Processed page numbers %i through %i" % (seg_from, seg_to - 1))


if __name__ == "__main__":
    t0 = mytime()  # start a timer
    filename = sys.argv[1]
    mat = fitz.Matrix(0.2, 0.2)  # the rendering matrix: scale down to 20%
    cpu = cpu_count()

    # make vectors of arguments for the processes
    vectors = [(i, cpu, filename, mat) for i in range(cpu)]
    print("Starting %i processes for '%s'." % (cpu, filename))

    pool = Pool()  # make pool of 'cpu_count()' processes
    pool.map(render_page, vectors, 1)  # start processes passing each a vector

    t1 = mytime()  # stop the timer
    print("Total time %g seconds" % round(t1 - t0, 2))
```

Here is a more complex example involving inter-process communication between a main process (showing a GUI) and a child process doing PyMuPDF access to a document.

```
"""
Created on 2019-05-01

@author: yinkaisheng@live.com
@copyright: 2019 yinkaisheng@live.com
@license: GNU GPL 3.0+

Demonstrate the use of multiprocessing with PyMuPDF
-----------------------------------------------------
This example shows some more advanced use of multiprocessing.
The main process show a Qt GUI and establishes a 2-way communication with
another process, which accesses a supported document.
"""
import os
import sys
import time
import multiprocessing as mp
```

(continues on next page)

(continued from previous page)

```
import queue
import fitz
from PyQt5 import QtCore, QtGui, QtWidgets

my_timer = time.clock if str is bytes else time.perf_counter


class DocForm(QtWidgets.QWidget):
    def __init__(self):
        super().__init__()
        self.process = None
        self.queNum = mp.Queue()
        self.queDoc = mp.Queue()
        self.pageCount = 0
        self.curPageNum = 0
        self.lastDir = ""
        self.timerSend = QtCore.QTimer(self)
        self.timerSend.timeout.connect(self.onTimerSendPageNum)
        self.timerGet = QtCore.QTimer(self)
        self.timerGet.timeout.connect(self.onTimerGetPage)
        self.timerWaiting = QtCore.QTimer(self)
        self.timerWaiting.timeout.connect(self.onTimerWaiting)
        self.initUI()

    def initUI(self):
        vbox = QtWidgets.QVBoxLayout()
        self.setLayout(vbox)

        hbox = QtWidgets.QHBoxLayout()
        self.btnOpen = QtWidgets.QPushButton("OpenDocument", self)
        self.btnOpen.clicked.connect(self.openDoc)
        hbox.addWidget(self.btnOpen)

        self.btnPlay = QtWidgets.QPushButton("PlayDocument", self)
        self.btnPlay.clicked.connect(self.playDoc)
        hbox.addWidget(self.btnPlay)

        self.btnStop = QtWidgets.QPushButton("Stop", self)
        self.btnStop.clicked.connect(self.stopPlay)
        hbox.addWidget(self.btnStop)

        self.label = QtWidgets.QLabel("0/0", self)
        self.label.setFont(QtGui.QFont("Verdana", 20))
        hbox.addWidget(self.label)

        vbox.addLayout(hbox)

        self.labelImg = QtWidgets.QLabel("Document", self)
        sizePolicy = QtWidgets.QSizePolicy(
            QtWidgets.QSizePolicy.Preferred, QtWidgets.QSizePolicy.Expanding
        )
        self.labelImg.setSizePolicy(sizePolicy)
        vbox.addWidget(self.labelImg)

        self.setGeometry(100, 100, 400, 600)
        self.setWindowTitle("PyMuPDF Document Player")
        self.show()
```

(continues on next page)

(continued from previous page)

```
    def openDoc(self):
        path, _ = QtWidgets.QFileDialog.getOpenFileName(
            self,
            "Open Document",
            self.lastDir,
            "All Supported Files (*.pdf;*.epub;*.xps;*.oxps;*.cbz;*.fb2);;PDF Files (*.pdf);;EPUB␣
↪Files (*.epub);;XPS Files (*.xps);;OpenXPS Files (*.oxps);;CBZ Files (*.cbz);;FB2 Files (*.fb2)",
            options=QtWidgets.QFileDialog.Options(),
        )
        if path:
            self.lastDir, self.file = os.path.split(path)
            if self.process:
                self.queNum.put(-1)  # use -1 to notify the process to exit
            self.timerSend.stop()
            self.curPageNum = 0
            self.pageCount = 0
            self.process = mp.Process(
                target=openDocInProcess, args=(path, self.queNum, self.queDoc)
            )
            self.process.start()
            self.timerGet.start(40)
            self.label.setText("0/0")
            self.queNum.put(0)
            self.startTime = time.perf_counter()
            self.timerWaiting.start(40)

    def playDoc(self):
        self.timerSend.start(500)

    def stopPlay(self):
        self.timerSend.stop()

    def onTimerSendPageNum(self):
        if self.curPageNum < self.pageCount - 1:
            self.queNum.put(self.curPageNum + 1)
        else:
            self.timerSend.stop()

    def onTimerGetPage(self):
        try:
            ret = self.queDoc.get(False)
            if isinstance(ret, int):
                self.timerWaiting.stop()
                self.pageCount = ret
                self.label.setText("{}/{}".format(self.curPageNum + 1, self.pageCount))
            else:  # tuple, pixmap info
                num, samples, width, height, stride, alpha = ret
                self.curPageNum = num
                self.label.setText("{}/{}".format(self.curPageNum + 1, self.pageCount))
                fmt = (
                    QtGui.QImage.Format_RGBA8888
                    if alpha
                    else QtGui.QImage.Format_RGB888
                )
                qimg = QtGui.QImage(samples, width, height, stride, fmt)
                self.labelImg.setPixmap(QtGui.QPixmap.fromImage(qimg))
```

(continues on next page)

(continued from previous page)

```
        except queue.Empty as ex:
            pass

    def onTimerWaiting(self):
        self.labelImg.setText(
            'Loading "{}", {:.2f}s'.format(
                self.file, time.perf_counter() - self.startTime
            )
        )

    def closeEvent(self, event):
        self.queNum.put(-1)
        event.accept()


def openDocInProcess(path, queNum, quePageInfo):
    start = my_timer()
    doc = fitz.open(path)
    end = my_timer()
    quePageInfo.put(doc.pageCount)
    while True:
        num = queNum.get()
        if num < 0:
            break
        page = doc.loadPage(num)
        pix = page.getPixmap()
        quePageInfo.put(
            (num, pix.samples, pix.width, pix.height, pix.stride, pix.alpha)
        )
    doc.close()
    print("process exit")


if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    form = DocForm()
    sys.exit(app.exec_())
```

4.6 General

4.6.1 How to Open with a Wrong File Extension

If you have a document with a wrong file extension for its type, you can still correctly open it.

Assume that “some.file” is actually an XPS. Open it like so:

```
>>> doc = fitz.open("some.file", filetype = "xps")
```

Note: MuPDF itself does not try to determine the file type from the file contents. **You** are responsible for supplying the filetype info in some way – either implicitely via the file extension, or explicitely as shown.

There are pure Python packages like filetype[50] that help you doing this. Also consult the *Document* chapter for a full description.

4.6.2 How to Embed or Attach Files

PDF supports incorporating arbitrary data. This can be done in one of two ways: "embedding" or "attaching". PyMuPDF supports both options.

1. Attached Files: data are **attached to a page** by way of a *FileAttachment* annotation with this statement: `annot = page.addFileAnnot(pos, ...)`, for details see *`Page.addFileAnnot()`*. The first parameter "pos" is the *Point*, where a "PushPin" icon should be placed on the page.
2. Embedded Files: data are embedded on the **document level** via method *`Document.embeddedFileAdd()`*.

The basic differences between these options are **(1)** you need edit permission to embed a file, but only annotation permission to attach, **(2)** like all annotations, attachments are visible on a page, embedded files are not.

There exist several example scripts: embedded-list.py[51], new-annots.py[52].

Also look at the sections above and at chapter *Appendix 3: Considerations on Embedded Files*.

4.6.3 How to Delete and Re-Arrange Pages

With PyMuPDF you have all options to copy, move, delete or re-arrange the pages of a PDF. Intuitive methods exist that allow you to do this on a page-by-page level, like the *`Document.copyPage()`* method.

Or you alternatively prepare a complete new page layout in form of a Python sequence, that contains the page numbers you want, in the sequence you want, and as many times as you want each page. The following may illustrate what can be done with *`Document.select()`*:

```
doc.select([1, 1, 1, 5, 4, 9, 9, 9, 0, 2, 2, 2])
```

Now let's prepare a PDF for double-sided printing (on a printer not directly supporting this):

The number of pages is given by `len(doc)` (equal to `doc.pageCount`). The following lists represent the even and the odd page numbers, respectively:

```
>>> p_even = [p in range(len(doc)) if p % 2 == 0]
>>> p_odd  = [p in range(len(doc)) if p % 2 == 1]
```

This snippet creates the respective sub documents which can then be used to print the document:

```
>>> doc.select(p_even)     # only the even pages left over
>>> doc.save("even.pdf")   # save the "even" PDF
>>> doc.close()            # recycle the file
>>> doc = fitz.open(doc.name) # re-open
>>> doc.select(p_odd)      # and do the same with the odd pages
>>> doc.save("odd.pdf")
```

[50] https://pypi.org/project/filetype/
[51] https://github.com/pymupdf/PyMuPDF/blob/master/examples/embedded-list.py
[52] https://github.com/pymupdf/PyMuPDF/blob/master/demo/new-annots.py

For more information also have a look at this Wiki article[53].

The following example will reverse the order of all pages (**extremely fast:** sub-second time for the 1310 pages of the *Adobe PDF Reference 1.7*):

```
>>> lastPage = len(doc) - 1
>>> for i in range(lastPage):
        doc.movePage(lastPage, i)  # move current last page to the front
```

This snippet duplicates the PDF with itself so that it will contain the pages `0, 1, ..., n, 0, 1, ..., n` **(extremely fast and without noticeably increasing the file size!)**:

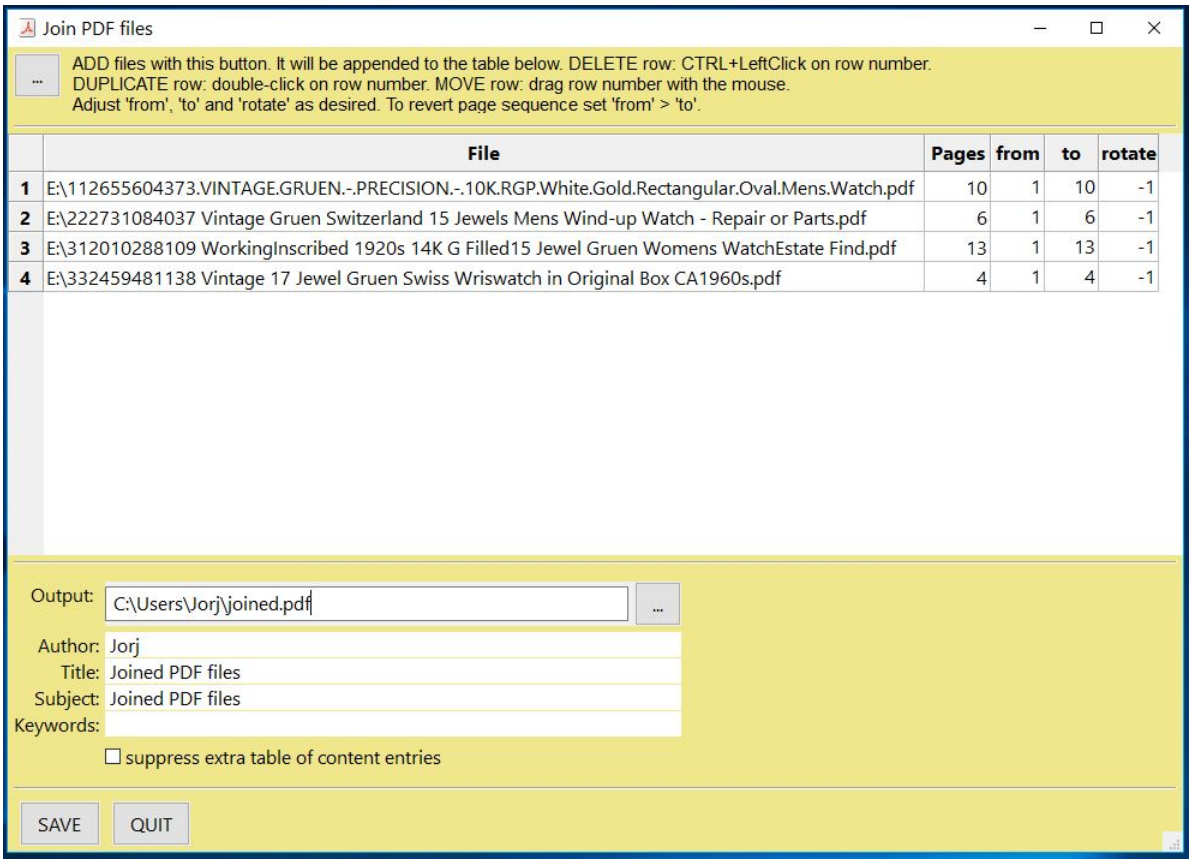
```
>>> pageCount = len(doc)
>>> for i in range(pageCount):
        doc.copyPage(i)  # copy this page to after last page
```

4.6.4 How to Join PDFs

It is easy to join PDFs with method *`Document.insertPDF()`*. Given open PDF documents, you can copy page ranges from one to the other. You can select the point where the copied pages should be placed, you can revert the page sequence and also change page rotation. This Wiki article[54] contains a full description.

The GUI script PDFjoiner.py[55] uses this method to join a list of files while also joining the respective table of contents segments. It looks like this:

[53] https://github.com/pymupdf/PyMuPDF/wiki/Rearranging-Pages-of-a-PDF
[54] https://github.com/pymupdf/PyMuPDF/wiki/Inserting-Pages-from-other-PDFs
[55] https://github.com/pymupdf/PyMuPDF/blob/master/examples/PDFjoiner.py



4.6.5 How to Add Pages

There two methods for adding new pages to a PDF: *Document.insertPage()* and *Document.newPage()* (and they share a common code base).

newPage

Document.newPage() returns the created *Page* object. Here is the constructor showing defaults:

```
>>> doc = fitz.open(...)              # some new or existing PDF document
>>> page = doc.newPage(to = -1,       # insertion point: end of document
                       width = 595,   # page dimension: A4 portrait
                       height = 842)
```

The above could also have been achieved with the short form `page = doc.newPage()`. The `to` parameter specifies the document's page number (0-based) **in front of which** to insert.

To create a page in *landscape* format, just exchange the width and height values.

Use this to create the page with another pre-defined paper format:

```
>>> w, h = fitz.PaperSize("letter-l")        # 'Letter' landscape
>>> page = doc.newPage(width = w, height = h)
```

The convenience function *PaperSize()* knows over 40 industry standard paper formats to choose from. To see them, inspect dictionary *paperSizes*. Pass the desired dictionary key to *PaperSize()* to retrieve

the paper dimensions. Upper and lower case is supported. If you append "-L" to the format name, the landscape version is returned.

Note: Here is a 3-liner that creates a PDF with one empty page. Its file size is 470 bytes:

```
>>> doc = fitz.open()
>>> doc.newPage()
>>> doc.save("A4.pdf")
```

---

insertPage

Document.insertPage() also inserts a new page and accepts the same parameters `to`, `width` and `height`. But it lets you also insert arbitrary text into the new page and returns the number of inserted lines:

```
>>> doc = fitz.open(...)              # some new or existing PDF document
>>> n = doc.insertPage(to = -1,       # default insertion point
                       text = None,   # string or sequence of strings
                       fontsize = 11,
                       width = 595,
                       height = 842,
                       fontname = "Helvetica", # default font
                       fontfile = None,        # any font file name
                       color = (0, 0, 0))      # text color (RGB)
```

The text parameter can be a (sequence of) string (assuming UTF-8 encoding). Insertion will start at *Point* (50, 72), which is one inch below top of page and 50 points from the left. The number of inserted text lines is returned. See the method definiton for more details.

4.6.6 How To Dynamically Clean Up Corrupt PDFs

This shows a potential use of PyMuPDF with another Python PDF library (the excellent pure Python package pdfrw[56] is used here as an example).

If a clean, non-corrupt / decompressed PDF is needed, one could dynamically invoke PyMuPDF to recover from many problems like so:

```
import sys
from io import BytesIO
from pdfrw import PdfReader
import fitz

#---------------------------------------
# 'Tolerant' PDF reader
#---------------------------------------
def reader(fname, password = None):
    idata = open(fname, "rb").read()        # read the PDF into memory and
    ibuffer = BytesIO(idata)                # convert to stream
    if password is None:
        try:
            return PdfReader(ibuffer)       # if this works: fine!
        except:
            pass
```

(continues on next page)

[56] https://pypi.python.org/pypi/pdfrw

(continued from previous page)

```
    # either we need a password or it is a problem-PDF
    # create a repaired / decompressed / decrypted version
    doc = fitz.open("pdf", ibuffer)
    if password is not None:                # decrypt if password provided
        rc = doc.authenticate(password)
        if not rc > 0:
            raise ValueError("wrong password")
    c = doc.write(garbage=3, deflate=True)
    del doc                                 # close & delete doc
    return PdfReader(BytesIO(c))            # let pdfrw retry
#---------------------------------------
# Main program
#---------------------------------------
pdf = reader("pymupdf.pdf", password = None) # inlude a password if necessary
print pdf.Info
# do further processing
```

With the command line utility `pdftk` (available[57] for Windows only, but reported to also run under Wine[58]) a similar result can be achieved, see here[59]. However, you must invoke it as a separate process via `subprocess.Popen`, using stdin and stdout as communication vehicles.

4.6.7 How to Split Single Pages

This deals with splitting up pages of a PDF in arbitrary pieces. For example, you may have a PDF with *Letter* format pages which you want to print with a magnification factor of four: each page is split up in 4 pieces which each go to a separate PDF page in *Letter* format again:

```
'''
Create a PDF copy with split-up pages (posterize)
-------------------------------------------------
License: GNU GPL V3
(c) 2018 Jorj X. McKie

Usage
------
python posterize.py input.pdf

Result
-------
A file "poster-input.pdf" with 4 output pages for every input page.

Notes
-----
(1) Output file is chosen to have page dimensions of 1/4 of input.

(2) Easily adapt the example to make n pages per input, or decide per each
    input page or whatever.

Dependencies
------------
```

(continues on next page)

[57] https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/
[58] https://www.winehq.org/
[59] http://www.overthere.co.uk/2013/07/22/improving-pypdf2-with-pdftk/

(continued from previous page)

```
PyMuPDF 1.12.2 or later
'''
from __future__ import print_function
import fitz, sys
infile = sys.argv[1]                          # input file name
src = fitz.open(infile)
doc = fitz.open()                             # empty output PDF

for spage in src:                             # for each page in input
    xref = 0                                  # force initial page copy to output
    r = spage.rect                            # input page rectangle
    d = fitz.Rect(spage.CropBoxPosition,      # CropBox displacement if not
                  spage.CropBoxPosition)      # starting at (0, 0)
    #---------------------------------------------------------------------
    # example: cut input page into 2 x 2 parts
    #---------------------------------------------------------------------
    r1 = r * 0.5                              # top left rect
    r2 = r1 + (r1.width, 0, r1.width, 0)      # top right rect
    r3 = r1 + (0, r1.height, 0, r1.height)    # bottom left rect
    r4 = fitz.Rect(r1.br, r.br)               # bottom right rect
    rect_list = [r1, r2, r3, r4]              # put them in a list

    for rx in rect_list:                      # run thru rect list
        rx += d                               # add the CropBox displacement
        page = doc.newPage(-1,                # new output page with rx dimensions
                           width = rx.width,
                           height = rx.height)
        page.showPDFpage(
                page.rect,  # fill all new page with the image
                src,        # input document
                spage.number, # input page number
                subrect = rx, # which part to use of input page
            )

# that's it, save output file
doc.save("poster-" + src.name,
         garbage = 3,                         # eliminate duplicate objects
         deflate = True)                      # compress stuff where possible
```

This shows what happens to an input page:

4.6.8 How to Combine Single Pages

This deals with joining PDF pages to form a new PDF with pages each combining two or four original ones (also called “2-up”, “4-up”, etc.). This could be used to create booklets or thumbnail-like overviews:

```
'''
Copy an input PDF to output combining every 4 pages
--------------------------------------------------
License: GNU GPL V3
(c) 2018 Jorj X. McKie

Usage
------
python 4up.py input.pdf

Result
-------
A file "4up-input.pdf" with 1 output page for every 4 input pages.

Notes
-----
(1) Output file is chosen to have A4 portrait pages. Input pages are scaled
    maintaining side proportions. Both can be changed, e.g. based on input
    page size. However, note that not all pages need to have the same size, etc.

(2) Easily adapt the example to combine just 2 pages (like for a booklet) or
    make the output page dimension dependent on input, or whatever.

Dependencies
-------------
PyMuPDF 1.12.1 or later
'''
from __future__ import print_function
import fitz, sys
infile = sys.argv[1]
src = fitz.open(infile)
doc = fitz.open()                       # empty output PDF

width, height = fitz.PaperSize("a4")    # A4 portrait output page format
r = fitz.Rect(0, 0, width, height)

# define the 4 rectangles per page
r1 = r * 0.5                            # top left rect
r2 = r1 + (r1.width, 0, r1.width, 0)    # top right
r3 = r1 + (0, r1.height, 0, r1.height) # bottom left
r4 = fitz.Rect(r1.br, r.br)             # bottom right

# put them in a list
r_tab = [r1, r2, r3, r4]

# now copy input pages to output
for spage in src:
    if spage.number % 4 == 0:           # create new output page
        page = doc.newPage(-1,
                           width = width,
                           height = height)
    # insert input page into the correct rectangle
    page.showPDFpage(r_tab[spage.number % 4],    # select output rect
                     src,                # input document
                     spage.number)       # input page number

# by all means, save new file using garbage collection and compression
doc.save("4up-" + infile, garbage = 3, deflate = True)
```

Example effect:

4.6.9 How to Convert Any Document to PDF

Here is a script that converts any PyMuPDF supported document to a PDF. These include XPS, EPUB, FB2, CBZ and all image formats, including multi-page TIFF images.

It features maintaining any metadata, table of contents and links contained in the source document:

```
from __future__ import print_function
"""
Demo script: Convert input file to a PDF
-----------------------------------------
Intended for multi-page input files like XPS, EPUB etc.

Features:
---------
Recovery of table of contents and links of input file.
While this works well for bookmarks (outlines, table of contents),
links will only work if they are not of type "LINK_NAMED".
This link type is skipped by the script.

For XPS and EPUB input, internal links however **are** of type "LINK_NAMED".
Base library MuPDF does not resolve them to page numbers.

So, for anyone expert enough to know the internal structure of these
document types, can further interpret and resolve these link types.

Dependencies
--------------
PyMuPDF v1.14.0+
"""
import sys
import fitz
if not (list(map(int, fitz.VersionBind.split(".")))) >= [1,14,0]):
    raise SystemExit("need PyMuPDF v1.14.0+")
fn = sys.argv[1]

print("Converting '%s' to '%s.pdf'" % (fn, fn))

doc = fitz.open(fn)

b = doc.convertToPDF()                      # convert to pdf
pdf = fitz.open("pdf", b)                   # open as pdf
```

(continues on next page)

(continued from previous page)

```
toc= doc.getToC()                           # table of contents of input
pdf.setToC(toc)                             # simply set it for output
meta = doc.metadata                         # read and set metadata
if not meta["producer"]:
    meta["producer"] = "PyMuPDF v" + fitz.VersionBind

if not meta["creator"]:
    meta["creator"] = "PyMuPDF PDF converter"
meta["modDate"] = fitz.getPDFnow()
meta["creationDate"] = meta["modDate"]
pdf.setMetadata(meta)

# now process the links
link_cnti = 0
link_skip = 0
for pinput in doc:                # iterate through input pages
    links = pinput.getLinks()     # get list of links
    link_cnti += len(links)       # count how many
    pout = pdf[pinput.number]     # read corresp. output page
    for l in links:               # iterate though the links
        if l["kind"] == fitz.LINK_NAMED:    # we do not handle named links
            print("named link page", pinput.number, l)
            link_skip += 1        # count them
            continue
        pout.insertLink(l)        # simply output the others

# save the conversion result
pdf.save(fn + ".pdf", garbage=4, deflate=True)
# say how many named links we skipped
if link_cnti > 0:
    print("Skipped %i named links of a total of %i in input." % (link_skip, link_cnti))
```

4.6.10 How to Deal with Messages Issued by MuPDF

Since PyMuPDF v1.16.0, error messages issued by the underlying MuPDF library are being redirected to the Python standard device `sys.stderr`. So you can handle them like any other output going to these devices.

We always prefix these messages with an identifying string `"mupdf:"`.

MuPDF warnings continue to be stored in an internal buffer and can be viewed using *Tools.mupdf_warnings()*. Please note that MuPDF errors may or may not lead to Python exceptions. In other words, you may see error messages from which MuPDF can recover and continue processing.

Example output for a **recoverable error**. We are opening a damaged PDF, but MuPDF is able to repair it and gives us a few information on what happened. Then we illustrate how to find out whether the document can later be saved incrementally:

```
>>> import fitz
>>> doc = fitz.open("damaged-file.pdf")  # leads to a sys.stderr message:
mupdf: cannot find startxref
>>> print(fitz.TOOLS.mupdf_warnings())  # check if there is more info:
trying to repair broken xref
```

(continues on next page)

(continued from previous page)

```
repairing PDF document
object missing 'endobj' token
>>> doc.can_save_incrementally()  # this is to be expected:
False
>>> # the document has nevertheless been created:
>>> doc
fitz.Document('damaged-file.pdf')
>>> # we now know that any save must occur to a new file
```

Example output for an **unrecoverable error**:

```
>>> import fitz
>>> doc = fitz.open("does-not-exist.pdf")
mupdf: cannot open does-not-exist.pdf: No such file or directory
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    doc = fitz.open("does-not-exist.pdf")
  File "C:\Users\Jorj\AppData\Local\Programs\Python\Python37\lib\site-packages\fitz\fitz.py", line␣
↪2200, in __init__
    _fitz.Document_swiginit(self, _fitz.new_Document(filename, stream, filetype, rect, width,␣
↪height, fontsize))
RuntimeError: cannot open does-not-exist.pdf: No such file or directory
>>>
```

4.6.11 How to Deal with PDF Encryption

Starting with version 1.16.0, PDF decryption and encryption (using passwords) are fully supported. You can do the following:

- Check whether a document is password protected / (still) encrypted (*Document.needsPass*, *Document.isEncrypted*).
- Gain access authorization to a document (*Document.authenticate()*).
- Set encryption details for PDF files using *Document.save()* or *Document.write()* and
  - decrypt or encrypt the content
  - set password(s)
  - set the encryption method
  - set permission details

Note: A PDF document may have two different passwords:

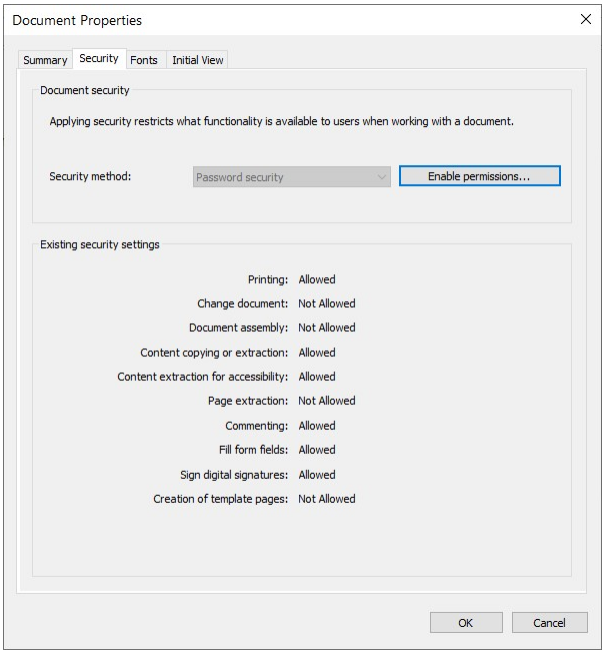
- The **owner password** provides full access rights, including changing passwords, encryption method, or permission detail.
- The **user password** provides access to document content according to the established permission details. If present, opening the PDF in a viewer will require providing it.

Method *Document.authenticate()* will automatically establish access rights according to the password used.

The following snippet creates a new PDF and encrypts it with separate user and owner passwords. Permissions are granted to print, copy and annotate, but no changes are allowed to someone authenticating with the user password:

```
import fitz

text = "some secret information"  # keep this data secret
perm = int(
    fitz.PDF_PERM_ACCESSIBILITY  # always use this
    | fitz.PDF_PERM_PRINT  # permit printing
    | fitz.PDF_PERM_COPY  # permit copying
    | fitz.PDF_PERM_ANNOTATE  # permit annotations
)
owner_pass = "owner"  # owner password
user_pass = "user"  # user password
encrypt_meth = fitz.PDF_ENCRYPT_AES_256  # strongest algorithm
doc = fitz.open()  # empty pdf
page = doc.newPage()  # empty page
page.insertText((50, 72), text)  # insert the data
doc.save(
    "secret.pdf",
    encryption=encrypt_meth,  # set the encryption method
    owner_pw=owner_pass,  # set the owner password
    user_pw=user_pass,  # set the user password
    permissions=perm,  # set permissions
)
```

Opening this document with some viewer (Nitro Reader 5) reflects these settings:



Decrypting will automatically happen on save as before when no encryption parameters are provided.

To **keep the encryption method** of a PDF save it using `encryption=fitz.PDF_ENCRYPT_KEEP`. If `doc.can_save_incrementally() == True`, an incremental save is also possible.

To **change the encryption method** specify the full range of options above (encryption, owner_pw, user_pw, permissions). An incremental save is **not possible** in this case.

4.7 Common Issues and their Solutions

4.7.1 Changing Annotations: Unexpected Behaviour

4.7.1.1 Problem

There are two scenarios:

1. Updating an annotation, which has been created by some other software, via a PyMuPDF script.
2. Creating an annotation with PyMuPDF and later changing it using some other PDF application.

In both cases you may experience unintended changes like a different annotation icon or text font, the fill color or line dashing have disappeared, line end symbols have changed their size or even have disappeared too, etc.

4.7.1.2 Cause

Annotation maintenance is handled differently by each PDF maintenance application (if it is supported at all). For any given PDF application, some annotation types may not be supported at all or only partly, or some details may be handled in a different way than with another application.

Almost always a PDF application also comes with its own icons (file attachments, sticky notes and stamps) and its own set of supported text fonts. For example:

- (Py-) MuPDF only supports these 5 basic fonts for 'FreeText' annotations: Helvetica, Times-Roman, Courier, ZapfDingbats and Symbol – no italics / no bold variations. When changing a 'FreeText' annotation created by some other app, its font will probably not be recognized nor accepted and be replaced by Helvetica.
- PyMuPDF fully supports the PDF text markers, but these types cannot be updated with Adobe Acrobat Reader.

In most cases there also exists limited support for line dashing which causes existing dashes to be replaced by straight lines. For example:

- PyMuPDF fully supports all line dashing forms, while other viewers only accept a limited subset.

4.7.1.3 Solutions

Unfortunately there is not much you can do in most of these cases.

1. Stay with the same software for **creating and changing** an annotation.
2. When using PyMuPDF to change an "alien" annotation, try to **avoid** *Annot.update()*. The following methods **can be used without it** so that the original appearance should be maintained:

- *Annot.setRect()* (location changes)
- *Annot.setFlags()* (annotation behaviour)
- *Annot.setInfo()* (meta information, except changes to `content`)
- *Annot.fileUpd()* (file attachment changes)

4.7.2 Misplaced Item Insertions on PDF Pages

4.7.2.1 Problem

You inserted an item (like an image, an annotation or some text) on an existing PDF page, but later you find it being placed at a different location than intended. For example an image should be inserted at the top, but it unexpectedly appears near the bottom of the page.

4.7.2.2 Cause

The creator of the PDF has established a non-standard page geometry without keeping it "local" (as they should!). Most commonly, the PDF standard point (0,0) at *bottom-left* has been changed to the *top-left* point. So top and bottom are reversed – causing your insertion to be misplaced.

The visible image of a PDF page is controlled by commands coded in a special mini-language. For an overview of this language consult "Operator Summary" on pp. 985 of the *Adobe PDF Reference 1.7*. These commands are stored in *contents* objects as strings (`bytes` in PyMuPDF).

There are commands in that language, which change the coordinate system of the page for all the following commands. In order to limit the scope of such commands local, they must be wrapped by the command pair `q` ("save graphics state", or "stack") and `Q` ("restore graphics state", or "unstack").

So the PDF creator did this:

```
stream
1 0 0 -1 0 792 cm    % <=== change of coordinate system:
...                  % letter page, top / bottom reversed
...                  % remains active beyond these lines
endstream
```

where they should have done this:

```
stream
q                    % put the following in a stack
1 0 0 -1 0 792 cm    % <=== scope of this is limited by Q command
...                  % here, a different geometry exists
Q                    % after this line, geometry of outer scope prevails
endstream
```

Note:

- In the mini-language's syntax, spaces and line breaks are equally accepted token delimiters.
- Multiple consecutive delimiters are treated as one.
- Keywords "stream" and "endstream" are inserted automatically – not by the programmer.

4.7.2.3 Solutions

Since v1.16.0, there is the property *Page._isWrapped*, which lets you check whether a page's contents are wrapped in that string pair.

If it is `False` or if you want to be on the safe side, pick one of the following:

1. The easiest way: in your script, do a *Page._cleanContents()* before you do your first item insertion.

2. Pre-process your PDF with the MuPDF command line utility `mutool clean -c ...` and work with its output file instead.
3. Directly wrap the page's *contents* with the stacking commands before you do your first item insertion.

Solutions 1. and 2. use the same technical basis and **do a lot more** than what is required in this context: they also clean up other inconsistencies or redundancies that may exist, multiple `/Contents` objects will be concatenated into one, and much more.

Note: For **incremental saves,** solution 1. has an unpleasant implication: it will bloat the update delta, because it changes so many things and, in addition, stores the **cleaned contents uncompressed**. So, if you use *Page._cleanContents()* you should consider **saving to a new file** with (at least) `garbage=3` and `deflate=True`.

Solution 3. is completely under your control and only does the minimum corrective action. There exists a handy low-level utility function which you can use for this. Suggested procedure:

- **Prepend** the missing stacking command by executing `fitz.TOOLS._insert_contents(page, b"q\n", False)`.
- **Append** an unstacking command by executing `fitz.TOOLS._insert_contents(page, b"\nQ", True)`.
- Alternatively, just use *Page._wrapContents()*, wich executes the previous two functions.

Note: If small incremental update deltas are a concern, this approach is the most effective. Other contents objects are not touched. The utility method creates two new PDF *stream* objects and inserts them before, resp. after the page's other *contents*. We therefore recommend the following snippet to get this situation under control:

```
>>> if not page._isWrapped:
        page._wrapContents()
>>> # start inserting text, images or annotations here
```

4.8 Low-Level Interfaces

Numerous methods are available to access and manipulate PDF files on a fairly low level. Admittedly, a clear distinction between "low level" and "normal" functionality is not always possible or subject to personal taste.

It also may happen, that functionality previously deemed low-level is lateron assessed as being part of the normal interface. This has happened in v1.14.0 for the class *Tools* – you now find it as an item in the Classes chapter.

Anyway – it is a matter of documentation only: in which chapter of the documentation do you find what. Everything is available always and always via the same interface.

4.8.1 How to Iterate through the xref Table

A PDF's *xref* table is a list of all objects defined in the file. This table may easily contain many thousand entries – the manual *Adobe PDF Reference 1.7* for example has over 330'000 objects. Table entry "0" is reserved and must not be touched. The following script loops through the *xref* table and prints each object's definition:

```
>>> xreflen = doc._getXrefLength()  # number of objects in file
>>> for xref in range(1, xreflen):  # skip item 0!
        print("")
        print("object %i (stream: %s)" % (xref, doc.isStream(xref)))
        print(doc._getXrefString(i, compressed=False))
```

This produces the following output:

```
object 1 (stream: False)
<<
    /ModDate (D:20170314122233-04'00')
    /PXCViewerInfo (PDF-XChange Viewer;2.5.312.1;Feb  9 2015;12:00:06;D:20170314122233-04'00')
>>

object 2 (stream: False)
<<
    /Type /Catalog
    /Pages 3 0 R
>>

object 3 (stream: False)
<<
    /Kids [ 4 0 R 5 0 R ]
    /Type /Pages
    /Count 2
>>

object 4 (stream: False)
<<
    /Type /Page
    /Annots [ 6 0 R ]
    /Parent 3 0 R
    /Contents 7 0 R
    /MediaBox [ 0 0 595 842 ]
    /Resources 8 0 R
>>
...
object 7 (stream: True)
<<
    /Length 494
    /Filter /FlateDecode
>>
...
```

A PDF object definition is an ordinary ASCII string.

4.8.2 How to Handle Object Streams

Some object types contain additional data apart from their object definition. Examples are images, fonts, embedded files or commands describing the appearance of a page.

Objects of these types are called "stream objects". PyMuPDF allows reading an object's stream via method *Document._getXrefStream()* with the object's *xref* as an argument. And it is also possible to write back a modified version of a stream using *Document._updateStream()*.

Assume that the following snippet wants to read all streams of a PDF for whatever reason:

```
>>> xreflen = doc._getXrefLength() # number of objects in file
>>> for xref in range(1, xreflen): # skip item 0!
        stream = doc._getXrefStream(xref)
        # do something with it (it is a bytes object or None)
        # e.g. just write it back:
        if stream:
            doc._updateStream(xref, stream)
```

Document._getXrefStream() automatically returns a stream decompressed as a bytes object – and *Document._updateStream()* automatically compresses it (where beneficial).

4.8.3 How to Handle Page Contents

A PDF page can have one or more *contents* objects – in fact, a page will be empty if it has no such object. These are stream objects describing **what** appears **where** on a page (like text and images). They are written in a special mini-language desribed e.g. in chapter "APPENDIX A - Operator Summary" on page 985 of the *Adobe PDF Reference 1.7*.

Every PDF reader application must be able to interpret the contents syntax to reproduce the intended appearance of the page.

If multiple *contents* objects are provided, they must be read and interpreted in the specified sequence in exactly the same way as if these streams were provided as a concatenation of the several.

There are good technical arguments for having multiple *contents* objects:

- It is a lot easier and faster to just add new *contents* objects than maintaining a single big one (which entails reading, decompressing, modifying, recompressing, and rewriting it for each change).
- When working with incremental updates, a modified big *contents* object will bloat the update delta and can thus easily negate the efficiency of incremental saves.

For example, PyMuPDF adds new, small *contents* objects in methods *Page.insertImage()*, *Page.showPDFpage()* and the *Shape* methods.

However, there are also situations when a **single** *contents* object is beneficial: it is easier to interpret and better compressible than multiple smaller ones.

Here are two ways of combining multiple contents of a page:

```
>>> # method 1: use the clean function
>>> for i in range(len(doc)):
        doc[i]._cleanContents() # cleans and combines multiple Contents
        page = doc[i]           # re-read the page (has only 1 contents now)
        cont = page._getContents()[0]
        # do something with the cleaned, combined contents
```

(continues on next page)

(continued from previous page)

```
>>> # method 2: concatenate multiple contents yourself
>>> for page in doc:
        cont = b""              # initialize contents
        for xref in page._getContents(): # loop through content xrefs
            cont += doc._getXrefStream(xref)
        # do something with the combined contents
```

The clean function *Page._cleanContents()* does a lot more than just glueing *contents* objects: it also corrects and optimizes the PDF operator syntax of the page and removes any inconsistencies.

4.8.4 How to Access the PDF Catalog

This is a central ("root") object of a PDF. It serves as a starting point to reach important other objects and it also contains some global options for the PDF:

```
>>> import fitz
>>> doc=fitz.open("PyMuPDF.pdf")
>>> cat = doc._getPDFroot()            # get xref of the /Catalog
>>> print(doc._getXrefString(cat))     # print object definition
<<
    /Type/Catalog                % object type
    /Pages 3593 0 R              % points to page tree
    /OpenAction 225 0 R          % action to perform on open
    /Names 3832 0 R              % points to global names tree
    /PageMode /UseOutlines       % initially show the TOC
    /PageLabels<</Nums[0<</S/D>>2<</S/r>>8<</S/D>>]>> % names given to pages
    /Outlines 3835 0 R           % points to outline tree
>>
```

Note: Indentation, line breaks and comments are inserted here for clarification purposes only and will not normally appear. For more information on the PDF catalog see section 3.6.1 on page 137 of the *Adobe PDF Reference 1.7*.

4.8.5 How to Access the PDF File Trailer

The trailer of a PDF file is a *dictionary* located towards the end of the file. It contains special objects, and pointers to important other information. See *Adobe PDF Reference 1.7* p. 96. Here is an overview:

Key	Type	Value
Size	int	Number of entries in the cross-reference table + 1.
Prev	int	Offset to previous *xref* section (indicates incremental updates).
Root	dictionary	(indirect) Pointer to the catalog. See previous section.
Encrypt	dictionary	Pointer to encryption object (encrypted files only).
Info	dictionary	(indirect) Pointer to information (metadata).
ID	array	File identifier consisting of two byte strings.
XRefStm	int	Offset of a cross-reference stream. See *Adobe PDF Reference 1.7* p. 109.

Access this information via PyMuPDF with *Document._getTrailerString()*.

```
>>> import fitz
>>> doc=fitz.open("PyMuPDF.pdf")
>>> trailer=doc._getTrailerString()
>>> print(trailer)
<</Size 5535/Info 5275 0 R/Root 5274 0 R/ID[(\340\273fE\225^l\226\232O|\003\201\325g\245)(}#1,
↪\317\205\000\371\251w06\352Oa\021)]>>
>>>
```

4.8.6 How to Access XML Metadata

A PDF may contain XML metadata in addition to the standard metadata format. In fact, most PDF reader or modification software adds this type of information when being used to save a PDF (Adobe, Nitro PDF, PDF-XChange, etc.).

PyMuPDF has no way to **interpret or change** this information directly, because it contains no XML features. The XML metadata is however stored as a *stream* object, so we do provide a way to **read the XML** stream and, potentially, also write back a modified stream or even delete it:

```
>>> metaxref = doc._getXmlMetadataXref()           # get xref of XML metadata
>>> # check if metaxref > 0!!!
>>> doc._getXrefString(metaxref)                  # object definition
'<</Subtype/XML/Length 3801/Type/Metadata>>'
>>> xmlmetadata = doc._getXrefStream(metaxref)    # XML data (stream - bytes obj)
>>> print(xmlmetadata.decode("utf8"))             # print str version of bytes
<?xpacket begin="\ufeff" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="3.1-702">
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
...
omitted data
...
<?xpacket end="w"?>
```

Using some XML package, the XML data can be interpreted and / or modified and then stored back:

```
>>> # write back modified XML metadata:
>>> doc._updateStream(metaxref, xmlmetadata)
>>>
>>> # if these data are not wanted, delete them:
>>> doc._delXmlMetadata()
```

CHAPTER

FIVE

CLASSES

5.1 Annot

This class is supported for PDF documents only.

Quote from the *Adobe PDF Reference 1.7*: "An annotation associates an object such as a note, sound, or movie with a location on a page of a PDF document, or provides a way to interact with the user by means of the mouse and keyboard."

There is a parent-child relationship between an annotation and its page. If the page object becomes unusable (closed document, any document structure change, etc.), then so does every of its existing annotation objects – an exception is raised saying that the object is "orphaned", whenever an annotation property or method is accessed.

Attribute	Short Description
Annot.fileGet()	return attached file content
Annot.fileInfo()	return attached file information
Annot.fileUpd()	set attached file new content
Annot.getPixmap()	image of the annotation as a pixmap
Annot.setBorder()	change the border
Annot.setColors()	change the colors
Annot.setFlags()	change the flags
Annot.setInfo()	change metadata
Annot.setLineEnds()	set line ending styles
Annot.setOpacity()	change transparency
Annot.setName()	change the "Name" field (e.g. icon name)
Annot.setRect()	change the rectangle
Annot.update()	apply accumulated annot changes
Annot.border	border details
Annot.colors	border / background and fill colors
Annot.flags	annotation flags
Annot.info	various information
Annot.lineEnds	start / end appearance of line-type annotations
Annot.next	link to the next annotation
Annot.opacity	the annot's transparency
Annot.parent	page object of the annotation
Annot.rect	rectangle containing the annotation
Annot.type	type of the annotation
Annot.vertices	point coordinates of Polygons, PolyLines, etc.
Annot.xref	the PDF *xref* number

Class API

class Annot

getPixmap(*matrix=fitz.Identity*, *colorspace=fitz.csRGB*, *alpha=False*)

Creates a pixmap from the annotation as it appears on the page in untransformed coordinates. The pixmap's *IRect* equals Annot.rect.irect (see below).

Parameters

- matrix (*Matrix*) – a matrix to be used for image creation. Default is the fitz.Identity matrix.
- colorspace (*Colorspace*) – a colorspace to be used for image creation. Default is fitz.csRGB.
- alpha (*bool*) – whether to include transparency information. Default is False.

Return type *Pixmap*

setInfo(*d*)

Changes the info dictionary. This includes dates, contents, subject and author (title). Changes for name will be ignored.

Parameters d (*dict*) – a dictionary compatible with the info property (see below). All entries must be strings.

setLineEnds(*start*, *end*)

Sets an annotation's line ending styles. Only 'FreeText', 'Line', 'PolyLine', and 'Polygon' annotations can have these properties. Each of these annotation types is defined by a list of points which are connected by lines. The symbol identified by start is attached to the first point, and end to the last point of this list. For unsupported annotation types, a no-operation with a warning message results.

Parameters

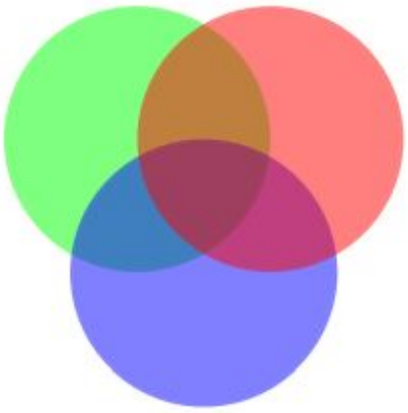
- start (*int*) – The symbol number for the first point.
- end (*int*) – The symbol number for the last point.

setOpacity(*value*)

Change an annotation's transparency.

Parameters value (*float*) – a float in range [0, 1]. Any value outside is assumed to be 1. E.g. a value of 0.5 sets the transparency to 50%.

Three overlapping 'Circle' annotations with each opacity set to 0.5:



setName(*name*)
New in version 1.16.0: Change the name field of any annotation type. For 'FileAttachment' and 'Text' annotations, this is the icon name, for 'Stamp' annotations the text in the stamp. The visual result (if any) depends on your PDF viewer. See also *Annotation Icons in MuPDF*.

Parameters `name` (*str*) – the new name.

setRect(*rect*)
Change the rectangle of an annotation. The annotation can be moved around and both sides of the rectangle can be independently scaled. However, the annotation appearance will never get rotated, flipped or sheared.

Parameters `rect` (*rect_like*) – the new rectangle of the annotation (finite and not empty). E.g. using a value of `annot.rect + (5, 5, 5, 5)` will shift the annot position 5 pixels to the right and downwards.

setBorder(*border*)
Change border width and dashing properties.

Parameters `border` (*dict*) – a dictionary with keys `"width"` (*float*), `"style"` (*str*) and `"dashes"` (*sequence*). Omitted keys will leave the resp. property unchanged. To e.g. remove dashing use: `"dashes": []`. If dashes is not an empty sequence, "style" will automatically set to "D" (dashed).

setFlags(*flags*)
Changes the annotation flags. Use the `|` operator to combine several.

Parameters `flags` (*int*) – an integer specifying the required flags.

setColors(*d*)
Changes the "stroke" and "fill" colors for supported annotation types.

Parameters `d` (*dict*) – a dictionary containing color specifications. For accepted dictionary keys and values see below. The most practical way should be to first make a copy of the `colors` property and then modify this dictionary as required.

update(*fontsize=0*, *text_color=None*, *border_color=None*, *fill_color=None*, *rotate=-1*)
Synchronize the appearance of an annotation with its properties after any changes.

You can safely omit this method only for the following changes:

- *setRect()*
- setFlasgs()
- *fileUpd()*

- *setInfo()* (except changes to "content")

All arguments are optional and **are reserved for 'FreeText'** annotations – because of implementation peculiarities of this annotation type. For other types they are ignored.

Color specifications may be made in the usual format used in PuMuPDF as sequences of floats ranging from 0.0 to 1.0 (including both). The sequence length must be 1, 3 or 4 (supporting GRAY, RGB and CMYK colorspaces respectively). For mono-color, just a float is also acceptable.

Parameters

- fontsize (*float*) – change font size of the text.
- text_color (*sequence*, *float*) – change the text color.
- border_color (*sequence*, *float*) – change the border color.
- fill_color (*sequence*, *float*) – the fill color. If you set (or leave) this to None, then **no rectangle at all** will be drawn around the text, and the border color will be ignored. This will leave anything "under" the text visible.
- rotate (*int*) – new rotation value. Default (-1) means no change.

Return type bool

fileInfo()
Basic information of the annot's attached file.

Return type dict

Returns a dictionary with keys filename, ufilename, desc (description), size (uncompressed file size), length (compressed length) for FileAttachment annot types, else None.

fileGet()
Returns attached file content.

Return type bytes

Returns the content of the attached file.

fileUpd(*buffer=None*, *filename=None*, *ufilename=None*, *desc=None*)
Updates the content of an attached file. All arguments are optional. No arguments lead to a no-op.

Parameters

- buffer (*bytes*/*bytearray*/*BytesIO*) – the new file content. Omit to only change meta-information.

  Changed in version 1.14.13: io.BytesIO is now also supported.
- filename (*str*) – new filename to associate with the file.
- ufilename (*str*) – new unicode filename to associate with the file.
- desc (*str*) – new description of the file content.

opacity
The annotation's transparency. If set, it is a value in range [0, 1]. The PDF default is 1.0. However, in an effort to tell the difference, we return -1.0 if not set.

Return type float

parent
The owning page object of the annotation.

Return type *Page*

`rect`
The rectangle containing the annotation.

Return type *Rect*

`next`
The next annotation on this page or `None`.

Return type `Annot`

`type`
A number and one or two strings describing the annotation type, like `[2, 'FreeText', 'FreeTextCallout']`. The second string entry is optional and may be empty. See the appendix *Annotation Related Constants* for a list of possible values and their meanings.

Return type list

`info`
A dictionary containing various information. All fields are (unicode) strings.

- `name` – e.g. for 'Stamp' annotations it will contain the stamp text like "Sold" or "Experimental", for other annot types you will see the name of the annot's icon here ("PushPin" for FileAttachment).
- `content` – a string containing the text for type `Text` and `FreeText` annotations. Commonly used for filling the text field of annotation pop-up windows.
- `title` – a string containing the title of the annotation pop-up window. By convention, this is used for the annotation author.
- `creationDate` – creation timestamp.
- `modDate` – last modified timestamp.
- `subject` – subject, an optional string.

Return type dict

`flags`
An integer whose low order bits contain flags for how the annotation should be presented.

Return type int

`lineEnds`
A pair of integers specifying start and end symbol of annotations types 'FreeText', 'Line', 'PolyLine', and 'Polygon'. `None` if not applicable. For possible values and descriptions in this list, see the *Adobe PDF Reference 1.7*, table 8.27 on page 630.

Return type tuple

`vertices`
A list containing a variable number of point ("vertices") coordinates (each given by a pair of floats) for various types of annotations:

- `Line` – the starting and ending coordinates (2 float pairs).
- `[2, 'FreeText', 'FreeTextCallout']` – 2 or 3 float pairs designating the starting, the (optional) knee point, and the ending coordinates.
- `PolyLine` / `Polygon` – the coordinates of the edges connected by line pieces (n float pairs for n points).

- text markup annotations – 4 float pairs specifying the `QuadPoints` of the marked text span (see *Adobe PDF Reference 1.7*, page 634).
- `Ink` – list of one to many sublists of vertex coordinates. Each such sublist represents a separate line in the drawing.

Return type list

`colors`
dictionary of two lists of floats in range `0 <= float <= 1` specifying the `stroke` and the interior (`fill`) colors. The stroke color is used for borders and everything that is actively painted or written ("stroked"). The fill color is used for the interior of objects like line ends, circles and squares. The lengths of these lists implicitely determine the colorspaces used: 1 = GRAY, 3 = RGB, 4 = CMYK. So `[1.0, 0.0, 0.0]` stands for RGB color red. Both lists can be `[]` if no color is specified. The value of each float `f` is mapped to the integer value `i` in range 0 to 255 via `f = i / 255`.

Return type dict

`xref`
The PDF *xref*.

Return type int

`border`
A dictionary containing border characteristics. Empty if no border information exists. The following keys may be present:

- `width` – a float indicating the border thickness in points. The value is -1.0 if no width is specified.
- `dashes` – a sequence of integers specifying a line dash pattern. `[]` means no dashes, `[n]` means equal on-off lengths of `n` points, longer lists will be interpreted as specifying alternating on-off length values. See the *Adobe PDF Reference 1.7* page 217 for more details.
- `style` – 1-byte border style: `S` (Solid) = solid rectangle surrounding the annotation, `D` (Dashed) = dashed rectangle surrounding the annotation, the dash pattern is specified by the `dashes` entry, `B` (Beveled) = a simulated embossed rectangle that appears to be raised above the surface of the page, `I` (Inset) = a simulated engraved rectangle that appears to be recessed below the surface of the page, `U` (Underline) = a single line along the bottom of the annotation rectangle.

Return type dict

5.1.1 Annotation Icons in MuPDF

This is a list of icons referencable by name for annotation types 'Text' and 'FileAttachment'. You can use them via the `icon` parameter when adding an annotation, or use the as argument in *Annot.setName()*. It is left to your discretion which item to choose when – no mechanism will keep you from using e.g. the "Speaker" icon for a 'FileAttachment'.

5.1.2 Example

Change the graphical image of an annotation. Also update the "author" and the text to be shown in the popup window:

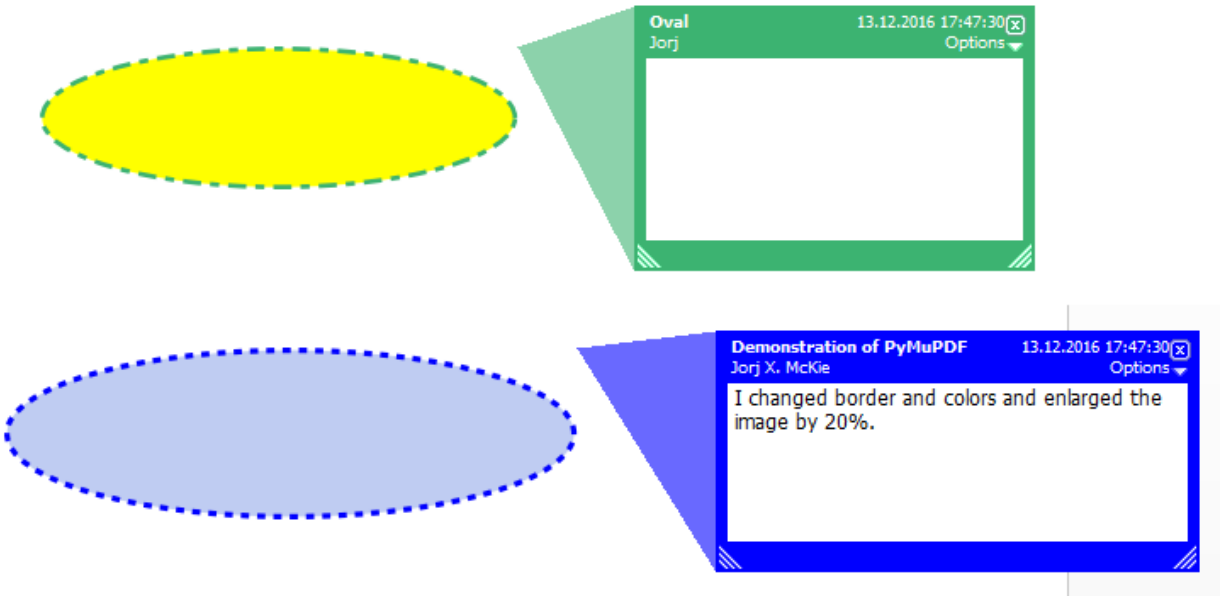
```
doc = fitz.open("circle-in.pdf")
page = doc[0]                           # page 0
annot = page.firstAnnot                 # get the annotation
annot.setBorder({"dashes": [3]})        # set dashes to "3 on, 3 off ..."

# set stroke and fill color to some blue
annot.setColors({"stroke":(0, 0, 1), "fill":(0.75, 0.8, 0.95)})
info = annot.info                       # get info dict
info["title"] = "Jorj X. McKie"         # set author

# text in popup window ...
info["content"] = "I changed border and colors and enlarged the image by 20%."
info["subject"] = "Demonstration of PyMuPDF"      # some PDF viewers also show this
annot.setInfo(info)                     # update info dict
r = annot.rect                          # take annot rect
r.x1 = r.x0 + r.width  * 1.2            # new location has same top-left
r.y1 = r.y0 + r.height * 1.2            # but 20% longer sides
annot.setRect(r)                        # update rectangle
```

(continues on next page)

(continued from previous page)

```
annot.update()                          # update the annot's appearance
doc.save("circle-out.pdf")              # save
```

This is how the circle annotation looks like before and after the change (pop-up windows displayed using Nitro PDF viewer):



5.2 Colorspace

Represents the color space of a *Pixmap*.

Class API

```
class Colorspace
```

`__init__`(*self*, *n*)
Constructor

Parameters `n` (*int*) – A number identifying the colorspace. Possible values are *CS_RGB*, *CS_GRAY* and *CS_CMYK*.

`name`
The name identifying the colorspace. Example: `fitz.csCMYK.name == 'DeviceCMYK'`.

Type str

`n`

The number of bytes required to define the color of one pixel. Example: `fitz.csCMYK.n == 4`.

type int

Predefined Colorspaces

For saving some typing effort, there exist predefined colorspace objects for the three available cases.

- *csRGB* = `fitz.Colorspace(fitz.CS_RGB)`
- *csGRAY* = `fitz.Colorspace(fitz.CS_GRAY)`
- *csCMYK* = `fitz.Colorspace(fitz.CS_CMYK)`

5.3 DisplayList

DisplayList is a list containing drawing commands (text, images, etc.). The intent is two-fold:

1. as a caching-mechanism to reduce parsing of a page
2. as a data structure in multi-threading setups, where one thread parses the page and another one renders pages. This aspect is currently not supported by PyMuPDF.

A `DisplayList` is populated with objects from a page usually by executing *Page.getDisplayList()*. There also exists an independent constructor.

"Replay" the list (once or many times) by invoking one of its methods *run()*, *getPixmap()* or *getTextPage()*.

Method	Short Description
run()	Run a display list through a device.
getPixmap()	generate a pixmap
getTextPage()	generate a text page
rect	mediabox of the display list

Class API

`class DisplayList`

`__init__`(*self*, *mediabox*)
Create a new display list.

Parameters `mediabox` (*Rect*) – The page's rectangle – output of `page.bound()`.

Return type `DisplayList`

`run`(*device*, *matrix*, *area*)
Run the display list through a device. The device will populate the display list with its "commands" (i.e. text extraction or image creation). The display list can later be used to "read" a page many times without having to re-interpret it from the document file.

You will most probably instead use one of the specialized run methods below – `getPixmap()` or `getTextPage()`.

Parameters

- `device` (*Device*) – Device
- `matrix` (*Matrix*) – Transformation matrix to apply to the display list contents.
- `area` (*Rect*) – Only the part visible within this area will be considered when the list is run through the device.

`getPixmap`(*matrix=fitz.Identity*, *colorspace=fitz.csRGB*, *alpha=0*, *clip=None*)
Run the display list through a draw device and return a pixmap.

Parameters

- `matrix` (*Matrix*) – matrix to use. Default is the identity matrix.
- `colorspace` (*Colorspace*) – the desired colorspace. Default is `RGB`.
- `alpha` (*int*) – determine whether or not (`0`, default) to include a transparency channel.
- `clip` (*IRect* or *Rect*) – an area of the full mediabox to which the pixmap should be restricted.

Return type *Pixmap*

Returns pixmap of the display list.

`getTextPage`(*flags*)

Run the display list through a text device and return a text page.

Parameters `flags` (*int*) – control which information is parsed into a text page. Default value in PyMuPDF is `3 = TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE`, i.e. ligatures are **passed through** ("æ" **will not be decomposed** into its components "a" and "e"), white spaces are **passed through** (not translated to spaces), and images are **not included**. See *Preserve Text Flags*.

Return type *TextPage*

Returns text page of the display list.

`rect`

Contains the display list's mediabox. This will equal the page's rectangle if it was created via `page.getDisplayList()`.

Type *Rect*

5.4 Document

This class represents a document. It can be constructed from a file or from memory.

Since version 1.9.0 there exists the alias `open` for this class.

For addional details on **embedded files** refer to Appendix 3.

Method / Attribute	Short Description
Document.authenticate()	gain access to an encrypted document
Document.can_save_incrementally()	check if incremental save is possible
Document.close()	close the document
Document.convertToPDF()	write a PDF version to memory
Document.copyPage()	PDF only: copy a page reference
Document.deletePage()	PDF only: delete a page
Document.deletePageRange()	PDF only: delete a page range
Document.embeddedFileAdd()	PDF only: add a new embedded file from buffer
Document.embeddedFileCount()	PDF only: number of embedded files
Document.embeddedFileDel()	PDF only: delete an embedded file entry
Document.embeddedFileGet()	PDF only: extract an embedded file buffer
Document.embeddedFileInfo()	PDF only: metadata of an embedded file

Continued on next page

Table 1 – continued from previous page

Method / Attribute	Short Description
Document.embeddedFileNames()	PDF only: list of embedded files
Document.embeddedFileUpd()	PDF only: change an embedded file
Document.fullcopyPage()	PDF only: duplicate a page
Document.getPageFontList()	PDF only: make a list of fonts on a page
Document.getPageImageList()	PDF only: make a list of images on a page
Document.getPagePixmap()	create a pixmap of a page by page number
Document.getPageText()	extract the text of a page by page number
Document.getSigFlags()	PDF only: determine signature state
Document.getToC()	create a table of contents
Document.insertPage()	PDF only: insert a new page
Document.insertPDF()	PDF only: insert pages from another PDF
Document.layout()	re-paginate the document (if supported)
Document.loadPage()	read a page
Document.movePage()	PDF only: move a page to another location
Document.newPage()	PDF only: insert a new empty page
Document.pages()	iterator over a page range
Document.save()	PDF only: save the document
Document.saveIncr()	PDF only: save the document incrementally
Document.searchPageFor()	search for a string on a page
Document.select()	PDF only: select a subset of pages
Document.setMetadata()	PDF only: set the metadata
Document.setToC()	PDF only: set the table of contents (TOC)
Document.write()	PDF only: writes the document to memory
Document.FormFonts	PDF only: list of global widget fonts
Document.isClosed	has document been closed?
Document.isEncrypted	document (still) encrypted?
Document.isFormPDF	is this a Form PDF?
Document.isPDF	is this a PDF?
Document.isReflowable	is this a reflowable document?
Document.metadata	metadata
Document.name	filename of document
Document.needsPass	require password to access data?
Document.outline	first *Outline* item
Document.pageCount	number of pages
Document.permissions	permissions to access the document

Class API

`class Document`

`__init__`(*self*, *filename=None*, *stream=None*, *filetype=None*, *rect=None*, *width=0*, *height=0*, *fontsize=11*)

Creates a `Document` object.

- With default parameters, a **new empty PDF** document will be created.
- If `stream` is given, then the document is created from memory and either `filename` or `filetype` must indicate its type.
- If `stream` is `None`, then a document is created from a file given by `filename`. Its type is inferred from the extension, which can be overruled by specifying `filetype`.

Parameters

- `filename` (*str*, *pathlib*) – A UTF-8 string or `pathlib` object containing a file path (or a file type, see below).
- `stream` (*bytes*, *bytearray*, *BytesIO*) – A memory area containing a supported document. Its type **must** be specified by either `filename` or `filetype`.

  Changed in version 1.14.13: `io.BytesIO` is now also supported.
- `filetype` (*str*) – A string specifying the type of document. This may be something looking like a filename (e.g. `"x.pdf"`), in which case MuPDF uses the extension to determine the type, or a mime type like `application/pdf`. Just using strings like `"pdf"` will also work.
- `rect` (*rect_like*) – a rectangle specifying the desired page size. This parameter is only meaningful for documents with a variable page layout ("reflowable" documents), like e-books or HTML, and ignored otherwise. If specified, it must be a non-empty, finite rectangle with top-left coordinates (0, 0). Together with parameter `fontsize`, each page will be accordingly laid out and hence also determine the number of pages.
- `width` (*float*) – may used together with `height` as an alternative to `rect` to specify layout information.
- `height` (*float*) – may used together with `width` as an alternative to `rect` to specify layout information.
- `fontsize` (*float*) – the default fontsize for reflowable document types. This parameter is ignored if none of the parameters `rect` or `width` and `height` are specified. Will be used to calculate the page layout.

Overview of possible forms (using the `open` synonym of `Document`):

```
>>> # from a file
>>> doc = fitz.open("some.pdf")
>>> doc = fitz.open("some.file", None, "pdf")  # copes with wrong extension
>>> doc = fitz.open("some.file", filetype="pdf")  # copes with wrong extension
```

```
>>> # from memory
>>> doc = fitz.open("pdf", mem_area)
>>> doc = fitz.open(None, mem_area, "pdf")
>>> doc = fitz.open(stream=mem_area, filetype="pdf")
```

```
>>> # new empty PDF
>>> doc = fitz.open()
```

`authenticate`(*password*)

Decrypts the document with the string `"password"`. If successful, document data can be accessed. For PDF documents, the "owner" and the "user" have different priviledges, and hence different passwords may exist for these authorization levels. The method will automatically establish the appropriate access rights for the provided password.

Parameters `password` (*str*) – owner or user password.

Return type int

Returns

a positive value if successful, zero otherwise. If successful, the indicator `isEncrypted` is set to `False`. Positive return codes carry the following information detail:

- bit 0 set => no password required – happens if method was used although *needsPass()* was zero.
- bit 1 set => **user** password authenticated
- bit 2 set => **owner** password authenticated

`loadPage(`*pno=0*`)`

Create a *Page* object for further processing (like rendering, text searching, etc.).

Parameters `pno` (*int*) – page number, zero-based (0 is default and the first page of the document). Any integer $-\infty < pno < pageCount$ is acceptable. If pno is negative, then *pageCount* will be added until this is no longer the case. For example: to load the last page, you can specify `doc.loadPage(-1)`. After this you have `page.number == doc.pageCount - 1`.

Return type *Page*

Note: Documents also follow the Python sequence protocol with page numbers as indices: `doc.loadPage(n) == doc[n]`. Consequently, expressions like `"for page in doc: ..."` and `"for page in reversed(doc): ..."` will successively yield the document's pages. Refer to `Document.range`()` which allows processing pages as with slicing.

`pages(`*start=None*[, *stop=None*[, *step=None*]])`

New in version 1.16.4: A generator for a given range of pages. Parameters have the same meaning as in the built-in function `"range()"`. Intended for expressions of the form `"for page in doc.pages(start, stop, step): ..."`.

Parameters

- `start` (*int*) – start iteration with this page number. Default is zero, allowed values are $-\infty < start < pageCount$. While this is negative, *pageCount* is added **before** starting the iteration.
- `stop` (*int*) – stop iteration at this page number. Default is *pageCount*, possible are $-\infty < stop \leq pageCount$. Larger values are **silently replaced** by the default. Negative values will cyclically emit the pages in reversed order. As with the built-in `"range()"`, this is the first page **not** returned.
- `step` (*int*) – stepping value. Defaults are 1 if start < stop and -1 if start > stop. Zero is not allowed.

Returns

a generator iterator over the document's pages. Some examples:

- `"for page in doc.pages()"` is the same as `"for page in doc"`.
- `"for page in doc.pages(4, 9, 2)"` emits pages 4, 6, 8.
- `"for page in doc.pages(0, None, 2)"` emits all pages with even numbers.
- `"doc.range(-2)"` emits the last two pages.
- `"for page in doc.pages(-1, -1)"` is the same as `"for page in reversed(doc)"`.

- `"for page in doc.pages(-1, -10)"` emits pages in reversed order, starting with the last document page **repeatedly**. For a 4-page document the following page numbers are emitted: 3, 2, 1, 0, 3, 2, 1, 0, 3, 2, 1, 0, 3.

convertToPDF(*from_page=-1*, *to_page=-1*, *rotate=0*)

Create a PDF version of the current document and write it to memory. **All document types** (except PDF) are supported. The parameters have the same meaning as in *insertPDF()*. In essence, you can restrict the conversion to a page subset, specify page rotation, and revert page sequence.

Parameters

- `from_page` (*int*) – first page to copy (0-based). Default is first page.
- `to_page` (*int*) – last page to copy (0-based). Default is last page.
- `rotate` (*int*) – rotation angle. Default is 0 (no rotation). Should be `n * 90` with an integer `n` (not checked).

Return type bytes

Returns a Python `bytes` object containing a PDF file image. It is created by internally using `write(garbage=4, deflate=True)`. See *write()*. You can output it directly to disk or open it as a PDF via `fitz.open("pdf", pdfbytes)`. Here are some examples:

```
>>> # convert an XPS file to PDF
>>> xps = fitz.open("some.xps")
>>> pdfbytes = xps.convertToPDF()
>>>
>>> # either do this --->
>>> pdf = fitz.open("pdf", pdfbytes)
>>> pdf.save("some.pdf")
>>>
>>> # or this --->
>>> pdfout = open("some.pdf", "wb")
>>> pdfout.write(pdfbytes)
>>> pdfout.close()
```

```
>>> # copy image files to PDF pages
>>> # each page will have image dimensions
>>> doc = fitz.open()                      # new PDF
>>> imglist = [ ... image file names ...] # e.g. a directory listing
>>> for img in imglist:
        imgdoc=fitz.open(img)            # open image as a document
        pdfbytes=imgdoc.convertToPDF()  # make a 1-page PDF of it
        imgpdf=fitz.open("pdf", pdfbytes)
        doc.insertPDF(imgpdf)            # insert the image PDF
>>> doc.save("allmyimages.pdf")
```

Note: The method uses the same logic as the `mutool convert` CLI. This works very well in most cases – however, beware of the following limitations.

- Image files: perfect, no issues detected. Apparently however, image transparency is ignored. If you need that (like for a watermark), use *Page.insertImage()* instead. Otherwise, this method is recommended for its much better prformance.
- XPS: appearance very good. Links work fine, outlines (bookmarks) are lost, but can easily

be recovered[67].

- EPUB, CBZ, FB2: similar to XPS.
- SVG: medium. Roughly comparable to svglib[60].

---

getToC(*simple=True*)

Creates a table of contents out of the document's outline chain.

Parameters `simple` (*bool*) – Indicates whether a simple or a detailed ToC is required. If `simple == False`, each entry of the list also contains a dictionary with *linkDest* details for each outline entry.

Return type list

Returns

a list of lists. Each entry has the form `[lvl, title, page, dest]`. Its entries have the following meanings:

- `lvl` – hierarchy level (positive *int*). The first entry is always 1. Entries in a row are either **equal**, **increase** by 1, or **decrease** by any number.
- `title` – title (*str*)
- `page` – 1-based page number (*int*). Page numbers `< 1` either indicate a target outside this document or no target at all (see next entry).
- `dest` – (*dict*) included only if `simple=False`. Contains details of the link destination.

getPagePixmap(*pno*, **args*, ***kwargs*)

Creates a pixmap from page `pno` (zero-based). Invokes *Page.getPixmap()*.

Parameters `pno` (*int*) – page number, 0-based in $-\infty < pno < pageCount$.

Return type *Pixmap*

getPageImageList(*pno*, *full=False*)

PDF only: Return a list of all image descriptions referenced by a page.

Parameters

- `pno` (*int*) – page number, 0-based in $-\infty < pno < pageCount$.
- `full` (*bool*) – whether to also include the *xref* of the *Form /XObject* where the item is referenced. This is zero if the item is part of page's /Resources.

Return type list

Returns

a list of images shown on this page. Each entry looks like `[xref, smask, width, height, bpc, colorspace, alt. colorspace, name, filter, form_xref]`. Where

- `xref` (*int*) is the image object number,
- `smask` (*int* optional) is the object number of its soft-mask image (if present),
- `width` and `height` (*ints*) are the image dimensions,

[67] However, you **can** use *Document.getToC()* and *Page.getLinks()* (which are available for all document types) and copy this information over to the output PDF. See demo pdf-converter.py[68].

[68] https://github.com/pymupdf/PyMuPDF/blob/master/demo/pdf-converter.py

[60] https://github.com/deeplook/svglib

- `bpc` (*int*) denotes the number of bits per component (a typical value is 8),
- `colorspace` (*str*)a string naming the colorspace (like `DeviceRGB`),
- `alt. colorspace` (*str* optional) is any alternate colorspace depending on the value of `colorspace`,
- `name` (*str*) is the symbolic name by which the **page references the image** in its content stream, and
- `filter` (*str* optional) is the decode filter of the image (*Adobe PDF Reference 1.7*, pp. 65).
- `form_xref` (*int* optional) the xref number of the *Form XObject*, which references the item. Zero if directly referenced by the page.

See below how this information can be used to extract PDF images as separate files. Another demonstration:

```
>>> doc = fitz.open("pymupdf.pdf")
>>> doc.getPageImageList(0, full=True)
[[316, 0, 261, 115, 8, 'DeviceRGB', '', 'Im1', 'DCTDecode', 0]]
>>> pix = fitz.Pixmap(doc, 316)  # 316 is the xref of the image
>>> pix
fitz.Pixmap(DeviceRGB, fitz.IRect(0, 0, 261, 115), 0)
```

Note: This list has no duplicate entries: the combination of *xref* and `name` is unique. But by themselves, each of the two may occur multiple times. The same image may well be referenced under different names within a page. Duplicate `name` entries on the other hand indicate the presence of "Form XObjects" on the page, e.g. generated by *Page.showPDFpage()*.

`getPageFontList`(*pno*, *full=False*)

PDF only: Return a list of all fonts referenced by the page.

Parameters

- `pno` (*int*) – page number, 0-based, any value < `len(doc)`.
- `full` (*bool*) – whether to also include the *xref* of the *Form /XObject* where the item is referenced. This is zero if the item is part of page's /Resources.

Return type list

Returns

a list of fonts referenced by this page. Each entry looks like `[xref, ext, type, basefont, name, encoding, form_xref]`. Where

- `xref` (*int*) is the font object number (may be zero if the PDF uses one of the builtin fonts directly),
- `ext` (*str*) font file extension (e.g. `ttf`, see *Font File Extensions*),
- `type` (*str*) is the font type (like `Type1` or `TrueType` etc.),
- `basefont` (*str*) is the base font name,
- `name` (*str*) is the reference name (or label), by which **the page references the font** in its contents stream(s), and
- `encoding` (*str* optional) the font's character encoding if different from its built-in encoding (*Adobe PDF Reference 1.7*, p. 414):

- `form_xref` (*int* optional) the xref number of the *Form XObject*, which references the item. Zero if directly referenced by the page.

```
>>> doc = fitz.open("some.pdf")
>>> for f in doc.getPageFontList(0, full=False): print(f)
[24, 'ttf', 'TrueType', 'DOKBTG+Calibri', 'R10', '']
[17, 'ttf', 'TrueType', 'NZNDCL+CourierNewPSMT', 'R14', '']
[32, 'ttf', 'TrueType', 'FNUUTH+Calibri-Bold', 'R8', '']
[28, 'ttf', 'TrueType', 'NOHSJV+Calibri-Light', 'R12', '']
[8, 'ttf', 'Type0', 'ECPLRU+Calibri', 'R23', 'Identity-H']
```

Note: This list has no duplicate entries: the combination of *xref* and `name` is unique. But by themselves, each of the two may occur multiple times. Duplicate `name` entries indicate the presence of "Form XObjects" on the page, e.g. generated by *Page.showPDFpage()*.

`getPageText`(*pno*, *output="text"*)

Extracts the text of a page given its page number `pno` (zero-based). Invokes *Page.getText()*.

Parameters

- `pno` (*int*) – page number, 0-based, any value `< len(doc)`.
- `output` (*str*) – A string specifying the requested output format: text, html, json or xml. Default is `text`.

Return type str

`layout`(*rect=None*, *width=0*, *height=0*, *fontsize=11*)

Re-paginate ("reflow") the document based on the given page dimension and fontsize. This only affects some document types like e-books and HTML. Ignored if not supported. Supported documents have `True` in property *isReflowable*.

Parameters

- `rect` (*rect_like*) – desired page size. Must be finite, not empty and start at point (0, 0).
- `width` (*float*) – use it together with `height` as alternative to `rect`.
- `height` (*float*) – use it together with `width` as alternative to `rect`.
- `fontsize` (*float*) – the desired default fontsize.

`select`(*s*)

PDF only: Keeps only those pages of the document whose numbers occur in the list. Empty sequences or elements outside `range(len(doc))` will cause a `ValueError`. For more details see remarks at the bottom or this chapter.

Parameters `s` (*sequence*) – The sequence (see *Using Python Sequences as Arguments in PyMuPDF*) of page numbers (zero-based) to be included. Pages not in the sequence will be deleted (from memory) and become unavailable until the document is reopened. **Page numbers can occur multiple times and in any order:** the resulting document will reflect the sequence exactly as specified.

Note:

- Page numbers in the sequence need not be unique nor be in any particular order. This makes the method a versatile utility to e.g. select only the even or the odd pages or meeting some other criteria and so forth.

- On a technical level, the method will always create a new *pagetree*.
- When dealing with only a few pages, methods *copyPage()*, *movePage()*, *deletePage()* are easier to use. In fact, they are also **much faster** – by at least one order of magnitude when the document has many pages.

---

setMetadata(*m*)

PDF only: Sets or updates the metadata of the document as specified in `m`, a Python dictionary. As with *select()*, these changes become permanent only when you save the document. Incremental save is supported.

Parameters `m` (*dict*) – A dictionary with the same keys as `metadata` (see below). All keys are optional. A PDF's format and encryption method cannot be set or changed and will be ignored. If any value should not contain data, do not specify its key or set the value to `None`. If you use `{}` all metadata information will be cleared to the string `"none"`. If you want to selectively change only some values, modify a copy of `doc.metadata` and use it as the argument. Arbitrary unicode values are possible if specified as UTF-8-encoded.

setToC(*toc*)

PDF only: Replaces the **complete current outline** tree (table of contents) with the new one provided as the argument. After successful execution, the new outline tree can be accessed as usual via method `getToC()` or via property `outline`. Like with other output-oriented methods, changes become permanent only via `save()` (incremental save supported). Internally, this method consists of the following two steps. For a demonstration see example below.

- Step 1 deletes all existing bookmarks.
- Step 2 creates a new TOC from the entries contained in `toc`.

Parameters `toc` (*sequence*) – A Python nested sequence with **all bookmark entries** that should form the new table of contents. Each entry is a list with the following format. Output variants of method `getToC()` are also acceptable as input.

- `[lvl, title, page, dest]`, where
  - `lvl` is the hierarchy level (int > 0) of the item, starting with `1` and being at most 1 higher than that of the predecessor,
  - `title` (str) is the title to be displayed. It is assumed to be UTF-8-encoded (relevant for multibyte code points only).
  - `page` (int) is the target page number **(attention: 1-based to support getToC()-output)**, must be in valid page range if positive. Set this to `-1` if there is no target, or the target is external.
  - `dest` (optional) is a dictionary or a number. If a number, it will be interpreted as the desired height (in points) this entry should point to on `page` in the current document. Use a dictionary (like the one given as output by `getToC(simple=False)`) if you want to store destinations that are either "named", or reside outside this documennt (other files, internet resources, etc.).

Return type int

Returns `outline` and `getToC()` will be updated upon successful execution. The return code will either equal the number of inserted items (`len(toc)`) or the number of deleted items if `toc` is an empty sequence.

Note: We currently always set the *Outline* attribute `is_open` to `False`. This shows all entries below level 1 as **collapsed**.

`can_save_incrementally()`
New in version 1.16.0: Check whether the document can be saved using option `incremental=True`. Use it to choose the right option without encountering exceptions.

`save(`*outfile*, *garbage=0*, *clean=False*, *deflate=False*, *incremental=False*, *ascii=False*, *expand=0*, *linear=False*, *pretty=False*, *encryption=PDF_ENCRYPT_NONE*, *permissions=-1*, *owner_pw=None*, *user_pw=None*`)`
PDF only: Saves the document in its **current state** under the name `outfile`.

Parameters

- `outfile` (*str*) – The file name to save to. Must be different from the original value if "incremental" is false or zero. When saving incrementally, "garbage" and "linear" **must be** false or zero and this parameter **must equal** the original filename (for convenience use `doc.name`).
- `garbage` (*int*) – Do garbage collection. Positive values exclude `incremental`.
  - 0 = none
  - 1 = remove unused objects
  - 2 = in addition to 1, compact the *xref* table
  - 3 = in addition to 2, merge duplicate objects
  - 4 = in addition to 3, check object streams for duplication (may be slow)
- `clean` (*bool*) – Clean and sanitize content streams[66]. Corresponds to `mutool clean` options `sc`.
- `deflate` (*bool*) – Deflate (compress) uncompressed streams.
- `incremental` (*bool*) – Only save changed objects. Excludes "garbage" and "linear". Cannot be used for files that are decrypted or repaired and also in some other cases. To be sure, check *Document.can_save_incrementally()*. If this is false, saving to a new file is required.
- `ascii` (*bool*) – Where possible convert binary data to ASCII.
- `expand` (*int*) – Decompress objects. Generates versions that can be better read by some other programs.
  - 0 = none
  - 1 = images
  - 2 = fonts
  - 255 = all
- `linear` (*bool*) – Save a linearised version of the document. This option creates a file format for improved performance when read via internet connections. Excludes "incremental".

[66] Content streams describe what (e.g. text or images) appears where and how on a page. PDF uses a specialized mini language similar to PostScript to do this (pp. 985 in *Adobe PDF Reference 1.7*), which gets interpreted when a page is loaded.

- pretty (*bool*) – Prettify the document source for better readability. PDF objects will be reformatted to look like the default output of *Document._getXrefString()*.
- permissions (*int*) – New in version 1.16.0: Set the desired permission levels. See *Document Permissions* for possible values. Default is granting all.
- encryption (*int*) – New in version 1.16.0: Set the desired encryption method. See *PDF encryption method codes* for possible values.
- owner_pw (*str*) – New in version 1.16.0: Set the document's owner password.
- user_pw (*str*) – New in version 1.16.0: Set the document's user password.

saveIncr()
PDF only: saves the document incrementally. This is a convenience abbreviation for `doc.save(doc.name, incremental=True, encryption=PDF_ENCRYPT_KEEP)`.

write(*garbage=0*, *clean=False*, *deflate=False*, *ascii=False*, *expand=0*, *linear=False*, *pretty=False*, *encryption=PDF_ENCRYPT_NONE*, *permissions=-1*, *owner_pw=None*, *user_pw=None*)
PDF only: Writes the **current content of the document** to a bytes object instead of to a file like `save()`. Obviously, you should be wary about memory requirements. The meanings of the parameters exactly equal those in *save()*. Chater *Collection of Recipes* contains an example for using this method as a pre-processor to pdfrw[61].

Changed in version 1.16.0.

Return type bytes

Returns a bytes object containing the complete document data.

searchPageFor(*pno*, *text*, *hit_max=16*, *quads=False*)
Search for `text` on page number `pno`. Works exactly like the corresponding *Page.searchFor()*. Any integer `-inf < pno < len(doc)` is acceptable.

insertPDF(*docsrc*, *from_page=-1*, *to_page=-1*, *start_at=-1*, *rotate=-1*, *links=True*, *annots=True*)
PDF only: Copy the page range **[from_page, to_page]** (including both) of PDF document `docsrc` into the current one. Inserts will start with page number `start_at`. Negative values can be used to indicate default values. All pages thus copied will be rotated as specified. Links can be excluded in the target, see below. All page numbers are zero-based.

Parameters

- docsrc (`Document`) – An opened PDF `Document` which must not be the current document object. However, it may refer to the same underlying file.
- from_page (*int*) – First page number in `docsrc`. Default is zero.
- to_page (*int*) – Last page number in `docsrc` to copy. Default is the last page.
- start_at (*int*) – First copied page will become page number `start_at` in the destination. If omitted, the page range will be appended to current document. If zero, the page range will be inserted before current first page.
- rotate (*int*) – All copied pages will be rotated by the provided value (degrees, integer multiple of 90).
- links (*bool*) – Choose whether (internal and external) links should be included in the copy. Default is `True`. An **internal link is always excluded**, if its destination is not one of the copied pages.

[61] https://pypi.python.org/pypi/pdfrw/0.3

- `annots` (*bool*) – New in version 1.16.1: Choose whether annotations should be included in the copy.

Note:

1. If `from_page > to_page`, pages will be **copied in reverse order**. If `0 <= from_page == to_page`, then one page will be copied.
2. `docsrc` bookmarks **will not be copied**. It is easy however, to recover a table of contents for the resulting document. Look at the examples below and at program PDFjoiner.py[62] in the *examples* directory: it can join PDF documents and at the same time piece together respective parts of the tables of contents.

`newPage`(*pno=-1*, *width=595*, *height=842*)

PDF only: Insert an empty page.

Parameters

- `pno` (*int*) – page number in front of which the new page should be inserted. Must be in `range(-1, len(doc) + 1)`. Special values -1 and `len(doc)` insert **after** the last page.
- `width` (*float*) – page width.
- `height` (*float*) – page height.

Return type *Page*

Returns the created page object.

`insertPage`(*pno*, *text=None*, *fontsize=11*, *width=595*, *height=842*, *fontname="helv"*, *fontfile=None*, *color=None*)

PDF only: Insert a new page and insert some text. Convenience function which combines *Document.newPage()* and (parts of) *Page.insertText()*.

Parameters `pno` (*int*) – page number (0-based) **in front of which** to insert. Must be in `range(-1, len(doc) + 1)`. Special values -1 and `len(doc)` insert **after** the last page.

Changed in version 1.14.12: This is now a positional parameter

For the other parameters, please consult the aforementioned methods.

Return type int

Returns the result of *Page.insertText()* (number of successfully inserted lines).

`deletePage`(*pno=-1*)

PDF only: Delete a page given by its 0-based number in $-\infty < pno < pageCount - 1$.

Changed in version 1.14.17.

Parameters `pno` (*int*) – the page to be deleted. Negative number count backwards from the end of the document (like with indices). Default is the last page.

`deletePageRange`(*from_page=-1*, *to_page=-1*)

PDF only: Delete a range of pages given as 0-based numbers. Any `-1` parameter will first be replaced by `len(doc) - 1` (ie. last page number). After that, condition `0 <= from_page <= to_page < len(doc)` must be true. If the parameters are equal, this is equivalent to *deletePage()*.

[62] https://github.com/pymupdf/PyMuPDF/blob/master/examples/PDFjoiner.py

Changed in version 1.14.17.

Parameters

- `from_page` (*int*) – the first page to be deleted.
- `to_page` (*int*) – the last page to be deleted.

Note: In an effort to maintain a valid PDF structure, this method and *deletePage()* will remove the deleted pages from the table of contents.

Similarly, it will **scan all pages** of the PDF and remove any links that point to deleted pages. Especially this action may have an extended response time for documents with a lot of pages.

In contrast, the **number of deleted pages** has a very small effect. So, whenever possible, you should delete page ranges instead of single pages.

Example:

```
>>> import time, fitz
>>> doc = fitz.open("Adobe PDF Reference 1-7.pdf")
>>> t0=time.perf_counter();doc.deletePageRange(500, 520);t1=time.perf_counter()
>>> round(t1 - t0, 2)
0.66
>>>
```

This is still more than 10 times faster than the corresponding *select()*:

```
>>> l = list(range(500)) + list(range(521, 1310))
>>> t0=time.perf_counter();doc.select(l);t1=time.perf_counter()
>>> round(t1 - t0, 2)
7.62
>>>
```

`copyPage`(*pno*, *to=-1*)

PDF only: Copy a page reference within the document.

Parameters

- `pno` (*int*) – the page to be copied. Must be in range `0 <= pno < len(doc)`.
- `to` (*int*) – the page number in front of which to copy. The default inserts **after** the last page.

Note: Only a new **reference** to the page object will be created – not a new page object, all copied pages will have identical attribute values, including the *Page.xref*. This implies that any changes to one of these copies will appear on all of them.

`fullcopyPage`(*pno*, *to=-1*)

New in version 1.14.17: PDF only: Make a new copy (duplicate) of a page.

Parameters

- `pno` (*int*) – the page to be duplicated. Must be in range `0 <= pno < len(doc)`.
- `to` (*int*) – the page number in front of which to copy. The default inserts **after** the last page.

Note: In contrast to *copyPage()*, this method creates a completely identical new page object – with the exception of *Page.xref* of course, which will be different. So changes to a copy will only show there.

movePage(*pno*, *to=-1*)

PDF only: Move (copy and then delete original) a page within the document.

Parameters

- pno (*int*) – the page to be moved. Must be in range `0 <= pno < len(doc)`.
- to (*int*) – the page number in front of which to insert the moved page. The default moves **after** the last page.

getSigFlags()

PDF only: Return whether the document contains signature fields.

Return type int

Returns

- -1: not a Form PDF or no signature fields exist.
- 1: at least one signature field exists.
- 3: contains signatures that may be invalidated if the file is saved (written) in a way that alters its previous contents, as opposed to an incremental update.

embeddedFileAdd(*name*, *buffer*, *filename=None*, *ufilename=None*, *desc=None*)

PDF only: Embed a new file. All string parameters except the name may be unicode (in previous versions, only ASCII worked correctly). File contents will be compressed (where beneficial).

Changed in version 1.14.16: The sequence of positional parameters "name" and "buffer" has been changed to comply with the layout of other functions.

Parameters

- name (*str*) – entry identifier, must not already exist.
- buffer (*bytes*, *bytearray*, *BytesIO*) – file contents.

  Changed in version 1.14.13: `io.BytesIO` is now also supported.
- filename (*str*) – optional filename. Documentation only, will be set to `name` if `None`.
- ufilename (*str*) – optional unicode filename. Documentation only, will be set to `filename` if `None`.
- desc (*str*) – optional description. Documentation only, will be set to `name` if `None`.

embeddedFileCount()

PDF only: Return the number of embedded files.

Changed in version 1.14.16: This is now a method. In previous versions, this was a property.

embeddedFileGet(*item*)

PDF only: Retrieve the content of embedded file by its entry number or name. If the document is not a PDF, or entry cannot be found, an exception is raised.

Parameters item (*int*, *str*) – index or name of entry. An integer must be in `range(embeddedFileCount())`.

Return type bytes

`embeddedFileDel(item)`

PDF only: Remove an entry from */EmbeddedFiles*. As always, physical deletion of the embedded file content (and file space regain) will occur only when the document is saved to a new file with a suitable garbage option.

Changed in version 1.14.16: Items can now be deleted by index, too.

Parameters `item` (*int/str*) – index or name of entry.

> **Warning:** When specifying an entry name, this function will only **delete the first item** with that name. Be aware that PDFs not created with PyMuPDF may contain duplicate names. So you may want to take appropriate precautions.

`embeddedFileInfo(item)`

PDF only: Retrieve information of an embedded file given by its number or by its name.

Parameters `item` (*int/str*) – index or name of entry. An integer must be in `range(embeddedFileCount())`.

Return type dict

Returns

a dictionary with the following keys:

- `name` – (*str*) name under which this entry is stored
- `filename` – (*str*) filename
- `ufilename` – (*unicode*) filename
- `desc` – (*str*) description
- `size` – (*int*) original file size
- `length` – (*int*) compressed file length

`embeddedFileNames()`

New in version 1.14.16: PDF only: Return a list of embedded file names. The sequence of names equals the physical sequence in the document.

Return type list

`embeddedFileUpd(item, buffer=None, filename=None, ufilename=None, desc=None)`

PDF only: Change an embedded file given its entry number or name. All parameters are optional. Letting them default leads to a no-operation.

Parameters

- `item` (*int/str*) – index or name of entry. An integer must be in `range(0, embeddedFileCount())`.
- `buffer` (*bytes , bytearray , BytesIO*) – the new file content.

  Changed in version 1.14.13: `io.BytesIO` is now also supported.
- `filename` (*str*) – the new filename.
- `ufilename` (*str*) – the new unicode filename.
- `desc` (*str*) – the new description.

embeddedFileSetInfo(*n*, *filename=None*, *ufilename=None*, *desc=None*)

PDF only: Change embedded file meta information. All parameters are optional. Letting them default will lead to a no-operation.

Parameters

- `n` (*int*, *str*) – index or name of entry. An integer must be in `range(embeddedFileCount())`.
- `filename` (*str*) – sets the filename.
- `ufilename` (*str*) – sets the unicode filename.
- `desc` (*str*) – sets the description.

Note: Deprecated subset of *embeddedFileUpd()*. Will be deleted in a future version.

---

close()

Release objects and space allocations associated with the document. If created from a file, also closes `filename` (releasing control to the OS).

outline

Contains the first *Outline* entry of the document (or `None`). Can be used as a starting point to walk through all outline items. Accessing this property for encrypted, not authenticated documents will raise an `AttributeError`.

Type *Outline*

isClosed

`False` if document is still open. If closed, most other attributes and methods will have been deleted / disabled. In addition, *Page* objects referring to this document (i.e. created with *Document.loadPage()*) and their dependent objects will no longer be usable. For reference purposes, *Document.name* still exists and will contain the filename of the original document (if applicable).

Type bool

isPDF

`True` if this is a PDF document, else `False`.

Type bool

isFormPDF

`False` if this is not a PDF or has no form fields, otherwise the number of root form fields (fields with no ancestors).

Changed in version 1.16.4: Returns the total number of (root) form fields.

Type bool,int

isReflowable

`True` if document has a variable page layout (like e-books or HTML). In this case you can set the desired page dimensions during document creation (open) or via method *layout()*.

Type bool

needsPass

Indicates whether the document is password-protected against access. This indicator remains unchanged – **even after the document has been authenticated**. Precludes incremental saves if true.

Type bool

isEncrypted
: This indicator initially equals needsPass. After successful authentication, it is set to False to reflect the situation.

  Type bool

permissions
: Contains the permissions to access the document. This is an integer containing bool values in respective bit positions. For example, if doc.permissions & fitz.PDF_PERM_MODIFY > 0, you may change the document. See *Document Permissions* for details.

  Changed in version 1.16.0::: This is now an integer comprised of bit indicators. Was a dictionary previously.

  Type int

metadata
: Contains the document's meta data as a Python dictionary or None (if isEncrypted=True and needPass=True). Keys are format, encryption, title, author, subject, keywords, creator, producer, creationDate, modDate. All item values are strings or None.

  Except format and encryption, for PDF documents, the key names correspond in an obvious way to the PDF keys /Creator, /Producer, /CreationDate, /ModDate, /Title, /Author, /Subject, and /Keywords respectively.

  - format contains the document format (e.g. 'PDF-1.6', 'XPS', 'EPUB').
  - encryption either contains None (no encryption), or a string naming an encryption method (e.g. 'Standard V4 R4 128-bit RC4'). Note that an encryption method may be specified **even if** needsPass=False. In such cases not all permissions will probably have been granted. Check *Document.permissions* for details.
  - If the date fields contain valid data (which need not be the case at all!), they are strings in the PDF-specific timestamp format "D:<TS><TZ>", where
    - <TS> is the 12 character ISO timestamp YYYYMMDDhhmmss (YYYY - year, MM - month, DD - day, hh - hour, mm - minute, ss - second), and
    - <TZ> is a time zone value (time intervall relative to GMT) containing a sign ('+' or '-'), the hour (hh), and the minute ('mm', note the apostrophies!).
  - A Paraguayan value might hence look like D:20150415131602-04'00', which corresponds to the timestamp April 15, 2015, at 1:16:02 pm local time Asuncion.

  Type dict

name
: Contains the filename or filetype value with which Document was created.

  Type str

pageCount
: Contains the number of pages of the document. May return 0 for documents with no pages. Function len(doc) will also deliver this result.

  Type int

FormFonts
: A list of form field font names defined in the /AcroForm object. None if not a PDF.

  Type list

Note: For methods that change the structure of a PDF (`insertPDF()`, `select()`, `copyPage()`, `deletePage()` and others), be aware that objects or properties in your program may have been invalidated or orphaned. Examples are *Page* objects and their children (links, annotations, widgets), variables holding old page counts, tables of content and the like. Remember to keep such variables up to date or delete orphaned objects. Also refer to *Ensuring Consistency of Important Objects in PyMuPDF*.

5.4.1 `setMetadata()` Example

Clear metadata information. If you do this out of privacy / data protection concerns, make sure you save the document as a new file with `garbage > 0`. Only then the old `/Info` object will also be physically removed from the file. In this case, you may also want to clear any XML metadata inserted by several PDF editors:

```
>>> import fitz
>>> doc=fitz.open("pymupdf.pdf")
>>> doc.metadata             # look at what we currently have
{'producer': 'rst2pdf, reportlab', 'format': 'PDF 1.4', 'encryption': None, 'author':
'Jorj X. McKie', 'modDate': "D:20160611145816-04'00'", 'keywords': 'PDF, XPS, EPUB, CBZ',
'title': 'The PyMuPDF Documentation', 'creationDate': "D:20160611145816-04'00'",
'creator': 'sphinx', 'subject': 'PyMuPDF 1.9.1'}
>>> doc.setMetadata({})      # clear all fields
>>> doc.metadata             # look again to show what happened
{'producer': 'none', 'format': 'PDF 1.4', 'encryption': None, 'author': 'none',
'modDate': 'none', 'keywords': 'none', 'title': 'none', 'creationDate': 'none',
'creator': 'none', 'subject': 'none'}
>>> doc._delXmlMetadata()    # clear any XML metadata
>>> doc.save("anonymous.pdf", garbage = 4)       # save anonymized doc
```

5.4.2 `setToC()` Demonstration

This shows how to modify or add a table of contents. Also have a look at csv2toc.py[63] and toc2csv.py[64] in the examples directory.

```
>>> import fitz
>>> doc = fitz.open("test.pdf")
>>> toc = doc.getToC()
>>> for t in toc: print(t)                          # show what we have
[1, 'The PyMuPDF Documentation', 1]
[2, 'Introduction', 1]
[3, 'Note on the Name fitz', 1]
[3, 'License', 1]
>>> toc[1][1] += " modified by setToC"              # modify something
>>> doc.setToC(toc)                                 # replace outline tree
3                                                   # number of bookmarks inserted
>>> for t in doc.getToC(): print(t)                 # demonstrate it worked
[1, 'The PyMuPDF Documentation', 1]
[2, 'Introduction modified by setToC', 1]           # <<< this has changed
[3, 'Note on the Name fitz', 1]
[3, 'License', 1]
```

[63] https://github.com/pymupdf/PyMuPDF/blob/master/examples/csv2toc.py
[64] https://github.com/pymupdf/PyMuPDF/blob/master/examples/toc2csv.py

5.4.3 insertPDF() Examples

(1) Concatenate two documents including their TOCs:

```
>>> doc1 = fitz.open("file1.pdf")          # must be a PDF
>>> doc2 = fitz.open("file2.pdf")          # must be a PDF
>>> pages1 = len(doc1)                     # save doc1's page count
>>> toc1 = doc1.getToC(False)      # save TOC 1
>>> toc2 = doc2.getToC(False)      # save TOC 2
>>> doc1.insertPDF(doc2)                   # doc2 at end of doc1
>>> for t in toc2:                         # increase toc2 page numbers
        t[2] += pages1                     # by old len(doc1)
>>> doc1.setToC(toc1 + toc2)               # now result has total TOC
```

Obviously, similar ways can be found in more general situations. Just make sure that hierarchy levels in a row do not increase by more than one. Inserting dummy bookmarks before and after `toc2` segments would heal such cases. A ready-to-use GUI (wxPython) solution can be found in script PDFjoiner.py[65] of the examples directory.

(2) More examples:

```
>>> # insert 5 pages of doc2, where its page 21 becomes page 15 in doc1
>>> doc1.insertPDF(doc2, from_page=21, to_page=25, start_at=15)
```

```
>>> # same example, but pages are rotated and copied in reverse order
>>> doc1.insertPDF(doc2, from_page=25, to_page=21, start_at=15, rotate=90)
```

```
>>> # put copied pages in front of doc1
>>> doc1.insertPDF(doc2, from_page=21, to_page=25, start_at=0)
```

5.4.4 Other Examples

Extract all page-referenced images of a PDF into separate PNG files:

```
for i in range(len(doc)):
    imglist = doc.getPageImageList(i)
    for img in imglist:
        xref = img[0]                   # xref number
        pix = fitz.Pixmap(doc, xref)    # make pixmap from image
        if pix.n - pix.alpha < 4:       # can be saved as PNG
            pix.writePNG("p%s-%s.png" % (i, xref))
        else:                           # CMYK: must convert first
            pix0 = fitz.Pixmap(fitz.csRGB, pix)
            pix0.writePNG("p%s-%s.png" % (i, xref))
            pix0 = None                 # free Pixmap resources
        pix = None                      # free Pixmap resources
```

Rotate all pages of a PDF:

```
>>> for page in doc: page.setRotation(90)
```

[65] https://github.com/pymupdf/PyMuPDF/blob/master/examples/PDFjoiner.py

5.5 Identity

Identity is a *Matrix* that performs no action – to be used whenever the syntax requires a matrix, but no actual transformation should take place. It has the form `fitz.Matrix(1, 0, 0, 1, 0, 0)`.

Identity is a constant, an "immutable" object. So, all of its matrix properties are read-only and its methods are disabled.

If you need a **mutable** identity matrix as a starting point, use one of the following statements:

```
>>> m = fitz.Matrix(1, 0, 0, 1, 0, 0)  # specify the values
>>> m = fitz.Matrix(1, 1)              # use scaling by factor 1
>>> m = fitz.Matrix(0)                 # use rotation by zero degrees
>>> m = fitz.Matrix(fitz.Identity)     # make a copy of Identity
```

5.6 IRect

IRect is a rectangular bounding box similar to *Rect*, except that all corner coordinates are integers. IRect is used to specify an area of pixels, e.g. to receive image data during rendering. Otherwise, many similarities exist, e.g. considerations concerning emptiness and finiteness of rectangles also apply to this class.

Attribute / Method	Short Description
IRect.contains()	checks containment of another object
IRect.getArea()	calculate rectangle area
IRect.getRect()	return a *Rect* with same coordinates
IRect.getRectArea()	calculate rectangle area
IRect.intersect()	common part with another rectangle
IRect.intersects()	checks for non-empty intersection
IRect.norm()	the Euclidean norm
IRect.normalize()	makes a rectangle finite
IRect.bottom_left	bottom left point, synonym `bl`
IRect.bottom_right	bottom right point, synonym `br`
IRect.height	height of the rectangle
IRect.isEmpty	whether rectangle is empty
IRect.isInfinite	whether rectangle is infinite
IRect.rect	equals result of method `getRect()`
IRect.top_left	top left point, synonym `tl`
IRect.top_right	top_right point, synonym `tr`
IRect.quad	*Quad* made from rectangle corners
IRect.width	width of the rectangle
IRect.x0	X-coordinate of the top left corner
IRect.x1	X-coordinate of the bottom right corner
IRect.y0	Y-coordinate of the top left corner
IRect.y1	Y-coordinate of the bottom right corner

Class API

`class IRect`

`__init__`(*self*)

`__init__`(*self*, *x0*, *y0*, *x1*, *y1*)

__init__(*self*, *irect*)

__init__(*self*, *sequence*)

Overloaded constructors. Also see examples below and those for the *Rect* class.

If another irect is specified, a **new copy** will be made.

If sequence is specified, it must be a Python sequence type of 4 numbers (see *Using Python Sequences as Arguments in PyMuPDF*). Non-integer numbers will be truncated, non-numeric entries will raise an exception.

The other parameters mean integer coordinates.

getRect()

A convenience function returning a *Rect* with the same coordinates. Also available as attribute `rect`.

Return type *Rect*

getRectArea([*unit*])

getArea([*unit*])

Calculates the area of the rectangle and, with no parameter, equals `abs(IRect)`. Like an empty rectangle, the area of an infinite rectangle is also zero.

Parameters `unit` (*str*) – Specify required unit: respective squares of "px" (pixels, default), "in" (inches), "cm" (centimeters), or "mm" (millimeters).

Return type float

intersect(*ir*)

The intersection (common rectangular area) of the current rectangle and `ir` is calculated and replaces the current rectangle. If either rectangle is empty, the result is also empty. If either rectangle is infinite, the other one is taken as the result – and hence also infinite if both rectangles were infinite.

Parameters `ir` (*rect_like*) – Second rectangle.

contains(*x*)

Checks whether `x` is contained in the rectangle. It may be *rect_like*, *point_like* or a number. If `x` is an empty rectangle, this is always true. Conversely, if the rectangle is empty this is always `False`, if `x` is not an empty rectangle and not a number. If `x` is a number, it will be checked to be one of the four components. `x in irect` and `irect.contains(x)` are equivalent.

Parameters `x` (*IRect* or *Rect* or *Point* or int) – the object to check.

Return type bool

intersects(*r*)

Checks whether the rectangle and the *rect_like* "r" contain a common non-empty *IRect*. This will always be `False` if either is infinite or empty.

Parameters `r` (*rect_like*) – the rectangle to check.

Return type bool

norm()

New in version 1.16.0: Return the Euclidean norm of the rectangle treated as a vector of four numbers.

normalize()

Make the rectangle finite. This is done by shuffling rectangle corners. After this, the bottom right corner will indeed be south-eastern to the top left one. See *Rect* for a more details.

`top_left`

`tl`
Equals `Point(x0, y0)`.

Type *Point*

`top_right`

`tr`
Equals `Point(x1, y0)`.

Type *Point*

`bottom_left`

`bl`
Equals `Point(x0, y1)`.

Type *Point*

`bottom_right`

`br`
Equals `Point(x1, y1)`.

Type *Point*

`quad`
The quadrilateral `Quad(irect.tl, irect.tr, irect.bl, irect.br)`.

Type *Quad*

`width`
Contains the width of the bounding box. Equals `abs(x1 - x0)`.

Type int

`height`
Contains the height of the bounding box. Equals `abs(y1 - y0)`.

Type int

`x0`
X-coordinate of the left corners.

Type int

`y0`
Y-coordinate of the top corners.

Type int

`x1`
X-coordinate of the right corners.

Type int

`y1`
Y-coordinate of the bottom corners.

Type int

`isInfinite`
`True` if rectangle is infinite, `False` otherwise.

Type bool

`isEmpty`
`True` if rectangle is empty, `False` otherwise.

Type bool

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
- Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.

5.7 Link

Represents a pointer to somewhere (this document, other documents, the internet). Links exist per document page, and they are forward-chained to each other, starting from an initial link which is accessible by the *Page.firstLink* property.

There is a parent-child relationship between a link and its page. If the page object becomes unusable (closed document, any document structure change, etc.), then so does every of its existing link objects – an exception is raised saying that the object is "orphaned", whenever a link property or method is accessed.

Attribute	Short Description
Link.setBorder()	modify border properties
Link.border	border characteristics
Link.colors	border line color
Link.dest	points to link destination details
Link.isExternal	external link destination?
Link.next	points to next link
Link.rect	clickable area in untransformed coordinates.
Link.uri	link destination
Link.xref	*xref* number of the entry

Class API

`class Link`

`setBorder(`*border*`)`
PDF only: Change border width and dashing properties.

Parameters `border` (*dict*) – a dictionary as returned by the *border* property, with keys `"width"` (*float*), `"style"` (*str*) and `"dashes"` (*sequence*). Omitted keys will leave the resp. property unchanged. To e.g. remove dashing use: `"dashes": []`. If dashes is not an empty sequence, "style" will automatically set to "D" (dashed).

`colors`
Meaningful for PDF only: A dictionary of two lists of floats in range `0 <= float <= 1` specifying the `stroke` and the interior (`fill`) colors. If not a PDF, `None` is returned. The stroke color is used for borders and everything that is actively painted or written ("stroked"). The lengths of these lists implicitely determine the colorspaces used: 1 = GRAY, 3 = RGB, 4 = CMYK. So `[1.0, 0.0, 0.0]` stands for RGB color red. Both lists can be `[]` if no color is specified. The value of each float `f` is mapped to the integer value `i` in range 0 to 255 via the computation `f = i / 255`.

Return type dict

`border`
Meaningful for PDF only: A dictionary containing border characteristics. It will be `None` for non-PDFs and an empty dictionary if no border information exists. The following keys can occur:

- `width` – a float indicating the border thickness in points. The value is -1.0 if no width is specified.
- `dashes` – a sequence of integers specifying a line dash pattern. `[]` means no dashes, `[n]` means equal on-off lengths of `n` points, longer lists will be interpreted as specifying alternating on-off length values. See the *Adobe PDF Reference 1.7* page 217 for more details.
- `style` – 1-byte border style: `S` (Solid) = solid rectangle surrounding the annotation, `D` (Dashed) = dashed rectangle surrounding the link, the dash pattern is specified by the `dashes` entry, `B` (Beveled) = a simulated embossed rectangle that appears to be raised above the surface of the page, `I` (Inset) = a simulated engraved rectangle that appears to be recessed below the surface of the page, `U` (Underline) = a single line along the bottom of the annotation rectangle.

Return type dict

`rect`
The area that can be clicked in untransformed coordinates.

Type *Rect*

`isExternal`
A bool specifying whether the link target is outside of the current document.

Type bool

`uri`
A string specifying the link target. The meaning of this property should be evaluated in conjunction with property `isExternal`. The value may be `None`, in which case `isExternal == False`. If `uri` starts with `file://`, `mailto:`, or an internet resource name, `isExternal` is `True`. In all other cases `isExternal == False` and `uri` points to an internal location. In case of PDF documents, this should either be `#nnnn` to indicate a 1-based (!) page number `nnnn`, or a named location. The format varies for other document types, e.g. `uri = '../FixedDoc.fdoc#PG_2_LNK_1'` for page number 2 (1-based) in an XPS document.

Type str

`xref`
An integer specifying the PDF *xref*. Zero if not a PDF.

Type int

`next`
The next link or `None`.

Type `Link`

`dest`
The link destination details object.

Type *linkDest*

5.8 linkDest

Class representing the *dest* property of an outline entry or a link. Describes the destination to which such entries point.

Attribute	Short Description
linkDest.dest	destination
linkDest.fileSpec	file specification (path, filename)
linkDest.flags	descriptive flags
linkDest.isMap	is this a MAP?
linkDest.isUri	is this a URI?
linkDest.kind	kind of destination
linkDest.lt	top left coordinates
linkDest.named	name if named destination
linkDest.newWindow	name of new window
linkDest.page	page number
linkDest.rb	bottom right coordinates
linkDest.uri	URI

Class API

```
class linkDest
```

`dest`
Target destination name if *linkDest.kind* is *LINK_GOTOR* and *linkDest.page* is -1.

Type str

`fileSpec`
Contains the filename and path this link points to, if *linkDest.kind* is *LINK_GOTOR* or *LINK_LAUNCH*.

Type str

`flags`
A bitfield describing the validity and meaning of the different aspects of the destination. As far as possible, link destinations are constructed such that e.g. *linkDest.lt* and *linkDest.rb* can be treated as defining a bounding box. But the flags indicate which of the values were actually specified, see *Link Destination Flags*.

Type int

`isMap`
This flag specifies whether to track the mouse position when the URI is resolved. Default value: False.

Type bool

`isUri`
Specifies whether this destination is an internet resource (as opposed to e.g. a local file specification in URI format).

Type bool

`kind`
Indicates the type of this destination, like a place in this document, a URI, a file launch, an action or a place in another file. Look at *Link Destination Kinds* to see the names and numerical values.

Type int

`lt`
The top left *Point* of the destination.

Type *Point*

`named`
This destination refers to some named action to perform (e.g. a javascript, see *Adobe PDF Reference 1.7*). Standard actions provided are `NextPage`, `PrevPage`, `FirstPage`, and `LastPage`.

Type str

`newWindow`
If true, the destination should be launched in a new window.

Type bool

`page`
The page number (in this or the target document) this destination points to. Only set if *`linkDest.kind`* is *`LINK_GOTOR`* or *`LINK_GOTO`*. May be `-1` if *`linkDest.kind`* is *`LINK_GOTOR`*. In this case *`linkDest.dest`* contains the **name** of a destination in the target document.

Type int

`rb`
The bottom right *Point* of this destination.

Type *Point*

`uri`
The name of the URI this destination points to.

Type str

5.9 Matrix

Matrix is a row-major 3x3 matrix used by image transformations in MuPDF (which complies with the respective concepts laid down in the *Adobe PDF Reference 1.7*). With matrices you can manipulate the rendered image of a page in a variety of ways: (parts of) the page can be rotated, zoomed, flipped, sheared and shifted by setting some or all of just six float values.

Since all points or pixels live in a two-dimensional space, one column vector of that matrix is a constant unit vector, and only the remaining six elements are used for manipulations. These six elements are usually represented by `[a, b, c, d, e, f]`. Here is how they are positioned in the matrix:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

Please note:

- the below methods are just convenience functions – everything they do, can also be achieved by directly manipulating the six numerical values

- all manipulations can be combined – you can construct a matrix that rotates **and** shears **and** scales **and** shifts, etc. in one go. If you however choose to do this, do have a look at the **remarks** further down or at the *Adobe PDF Reference 1.7*.

Method / Attribute	Description
Matrix.preRotate()	perform a rotation
Matrix.preScale()	perform a scaling
Matrix.preShear()	perform a shearing (skewing)
Matrix.preTranslate()	perform a translation (shifting)
Matrix.concat()	perform a matrix multiplication
Matrix.invert()	calculate the inverted matrix
Matrix.norm()	the Euclidean norm
Matrix.a	zoom factor X direction
Matrix.b	shearing effect Y direction
Matrix.c	shearing effect X direction
Matrix.d	zoom factor Y direction
Matrix.e	horizontal shift
Matrix.f	vertical shift
Matrix.isRectilinear	true if rect corners will remain rect corners

Class API

`class Matrix`

`__init__`(*self*)

`__init__`(*self*, *zoom-x*, *zoom-y*)

`__init__`(*self*, *shear-x*, *shear-y*, *1*)

`__init__`(*self*, *a*, *b*, *c*, *d*, *e*, *f*)

`__init__`(*self*, *matrix*)

`__init__`(*self*, *degree*)

`__init__`(*self*, *sequence*)

Overloaded constructors.

Without parameters, the zero matrix `Matrix(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)` will be created.

`zoom-*` and `shear-*` specify zoom or shear values (float) and create a zoom or shear matrix, respectively.

For "matrix" a **new copy** of another matrix will be made.

Float value "degree" specifies the creation of a rotation matrix which rotates anit-clockwise.

A "sequence" must be any Python sequence object with exactly 6 float entries (see *Using Python Sequences as Arguments in PyMuPDF*).

`fitz.Matrix(1, 1)`, `fitz.Matrix(0.0)` and `fitz.Matrix(fitz.Identity)` create modifyable versions of the *Identity* matrix, which looks like `[1, 0, 0, 1, 0, 0]`.

`norm()`

New in version 1.16.0: Return the Euclidean norm of the matrix as a vector.

`preRotate`(*deg*)

Modify the matrix to perform a counter-clockwise rotation for positive `deg` degrees, else clockwise. The matrix elements of an identity matrix will change in the following way:

`[1, 0, 0, 1, 0, 0] -> [cos(deg), sin(deg), -sin(deg), cos(deg), 0, 0]`.

Parameters `deg` (*float*) – The rotation angle in degrees (use conventional notation based on Pi = 180 degrees).

`preScale`(*sx*, *sy*)

Modify the matrix to scale by the zoom factors sx and sy. Has effects on attributes `a` thru `d` only: `[a, b, c, d, e, f] -> [a*sx, b*sx, c*sy, d*sy, e, f]`.

Parameters

- `sx` (*float*) – Zoom factor in X direction. For the effect see description of attribute `a`.
- `sy` (*float*) – Zoom factor in Y direction. For the effect see description of attribute `d`.

`preShear`(*sx*, *sy*)

Modify the matrix to perform a shearing, i.e. transformation of rectangles into parallelograms (rhomboids). Has effects on attributes `a` thru `d` only: `[a, b, c, d, e, f] -> [c*sy, d*sy, a*sx, b*sx, e, f]`.

Parameters

- `sx` (*float*) – Shearing effect in X direction. See attribute `c`.
- `sy` (*float*) – Shearing effect in Y direction. See attribute `b`.

`preTranslate`(*tx*, *ty*)

Modify the matrix to perform a shifting / translation operation along the x and / or y axis. Has effects on attributes `e` and `f` only: `[a, b, c, d, e, f] -> [a, b, c, d, tx*a + ty*c, tx*b + ty*d]`.

Parameters

- `tx` (*float*) – Translation effect in X direction. See attribute `e`.
- `ty` (*float*) – Translation effect in Y direction. See attribute `f`.

`concat`(*m1*, *m2*)

Calculate the matrix product `m1 * m2` and store the result in the current matrix. Any of `m1` or `m2` may be the current matrix. Be aware that matrix multiplication is not commutative. So the sequence of `m1`, `m2` is important.

Parameters

- `m1` (*Matrix*) – First (left) matrix.
- `m2` (*Matrix*) – Second (right) matrix.

`invert`(*m = None*)

Calculate the matrix inverse of `m` and store the result in the current matrix. Returns `1` if `m` is not invertible ("degenerate"). In this case the current matrix **will not change**. Returns `0` if `m` is invertible, and the current matrix is replaced with the inverted `m`.

Parameters `m` (*Matrix*) – Matrix to be inverted. If not provided, the current matrix will be used.

Return type int

`a`

Scaling in X-direction **(width)**. For example, a value of 0.5 performs a shrink of the **width** by a factor of 2. If a < 0, a left-right flip will (additionally) occur.

Type float

`b`
: Causes a shearing effect: each `Point(x, y)` will become `Point(x, y - b*x)`. Therefore, looking from left to right, e.g. horizontal lines will be "tilt" – downwards if b > 0, upwards otherwise (b is the tangens of the tilting angle).

  Type float

`c`
: Causes a shearing effect: each `Point(x, y)` will become `Point(x - c*y, y)`. Therefore, looking upwards, vertical lines will be "tilt" – to the left if c > 0, to the right otherwise (c ist the tangens of the tilting angle).

  Type float

`d`
: Scaling in Y-direction **(height)**. For example, a value of 1.5 performs a stretch of the **height** by 50%. If d < 0, an up-down flip will (additionally) occur.

  Type float

`e`
: Causes a horizontal shift effect: Each `Point(x, y)` will become `Point(x + e, y)`. Positive (negative) values of `e` will shift right (left).

  Type float

`f`
: Causes a vertical shift effect: Each `Point(x, y)` will become `Point(x, y - f)`. Positive (negative) values of `f` will shift down (up).

  Type float

`isRectilinear`
: Rectilinear means that no shearing is present and that any rotations are integer multiples of 90 degrees. Usually this is used to confirm that (axis-aligned) rectangles before the transformation are still axis-aligned rectangles afterwards.

  Type bool

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
- A matrix can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.
- Changes of matrix properties and execution of matrix methods can be executed consecutively. This is the same as multiplying the respective matrices.
- Matrix multiplication is **not commutative** – changing the execution sequence in general changes the result. So it can quickly become unclear which result a transformation will yield.

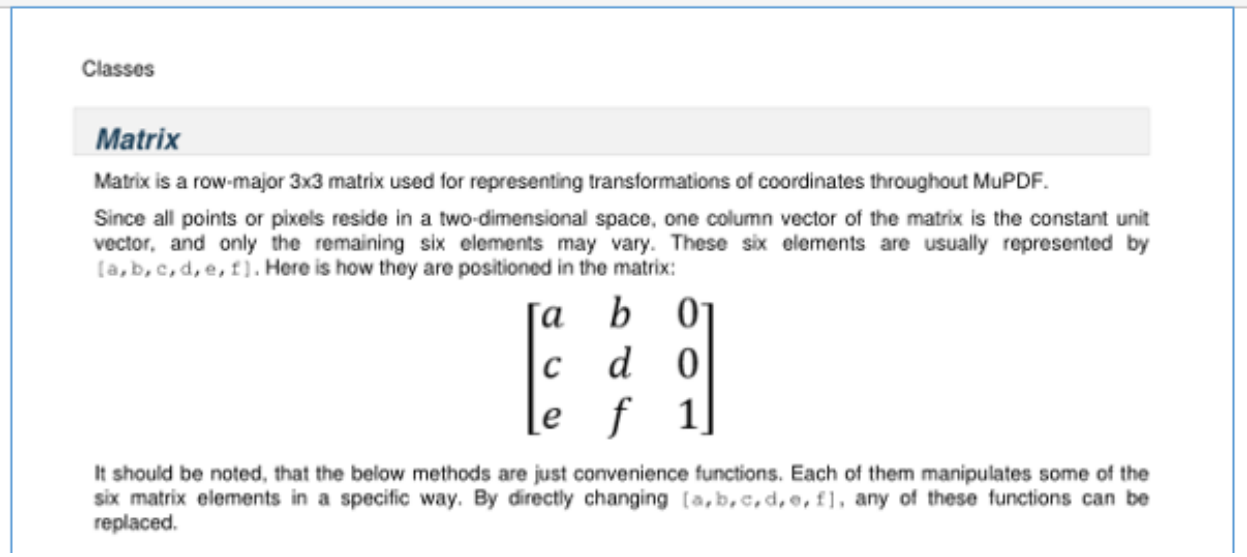
To keep results foreseeable for a series of matrix operations, Adobe recommends the following approach (*Adobe PDF Reference 1.7*, page 206):

1. Shift ("translate")
2. Rotate
3. Scale or shear ("skew")

5.9.1 Examples

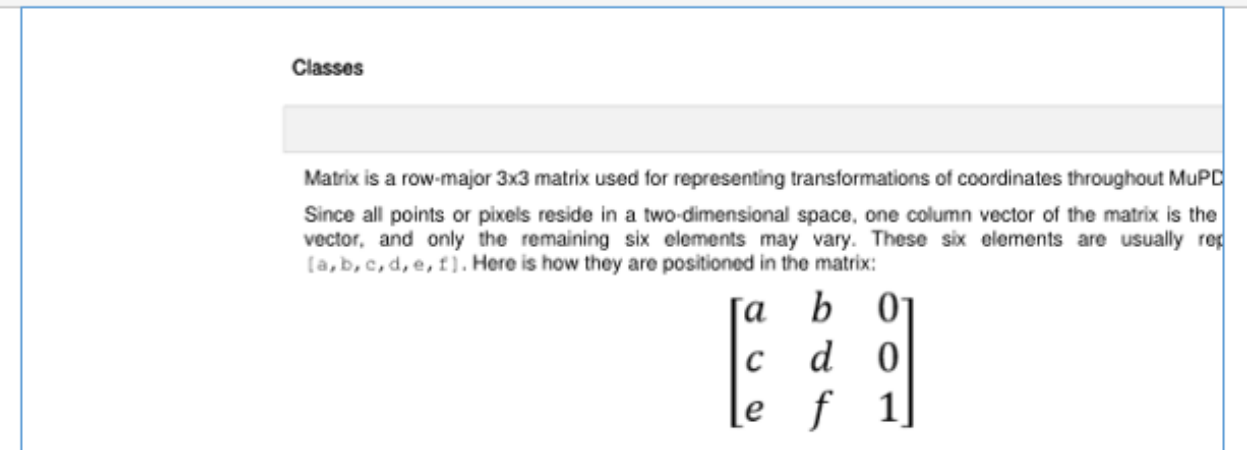
Here are examples to illustrate some of the effects achievable. The following pictures start with a page of the PDF version of this help file. We show what happens when a matrix is being applied (though always full pages are created, only parts are displayed here to save space).

This is the original page image:

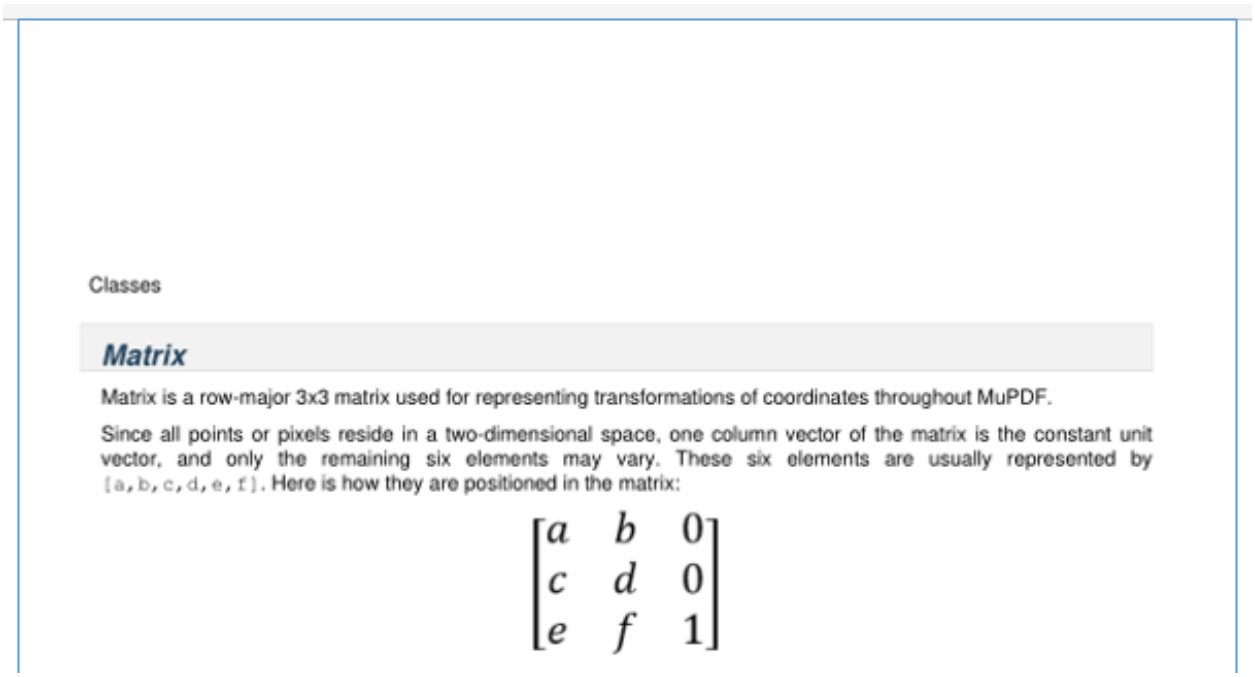


5.9.2 Shifting

We transform it with a matrix where `e = 100` (right shift by 100 pixels).



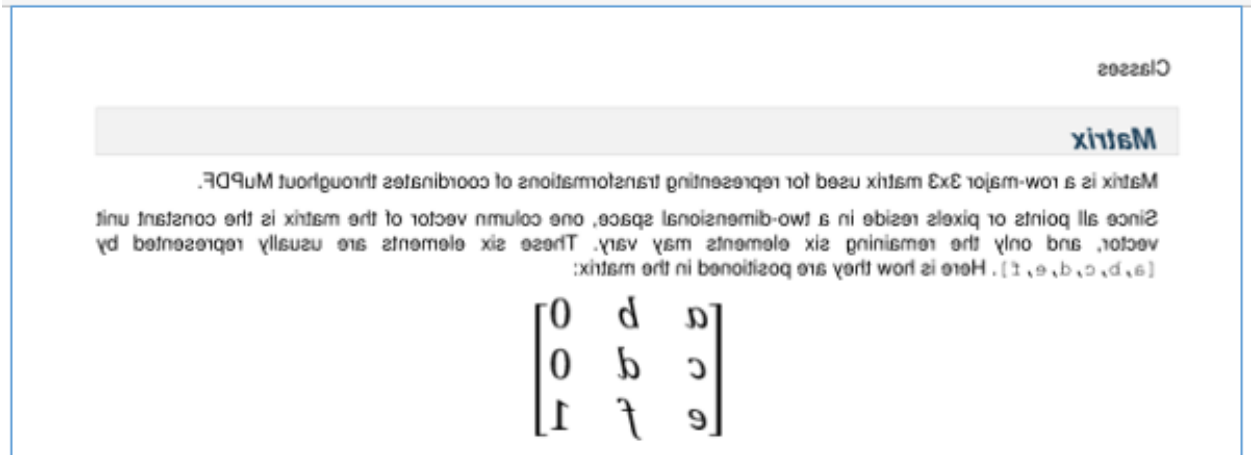
Next we do a down shift by 100 pixels: `f = 100`.



$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

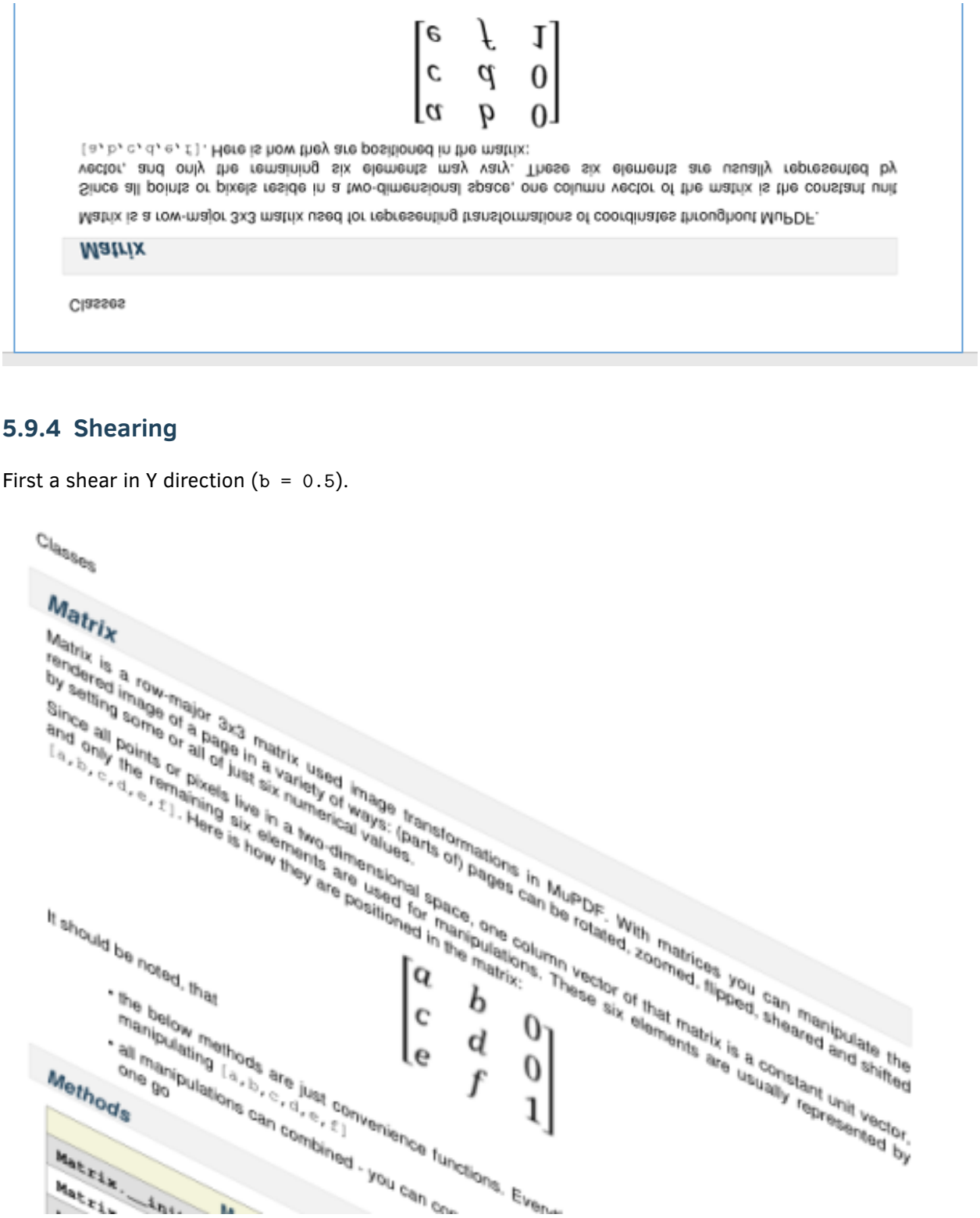
5.9.3 Flipping

Flip the page left-right (`a = -1`).



$$\begin{bmatrix} 0 & b & a \\ 0 & d & c \\ 1 & f & e \end{bmatrix}$$

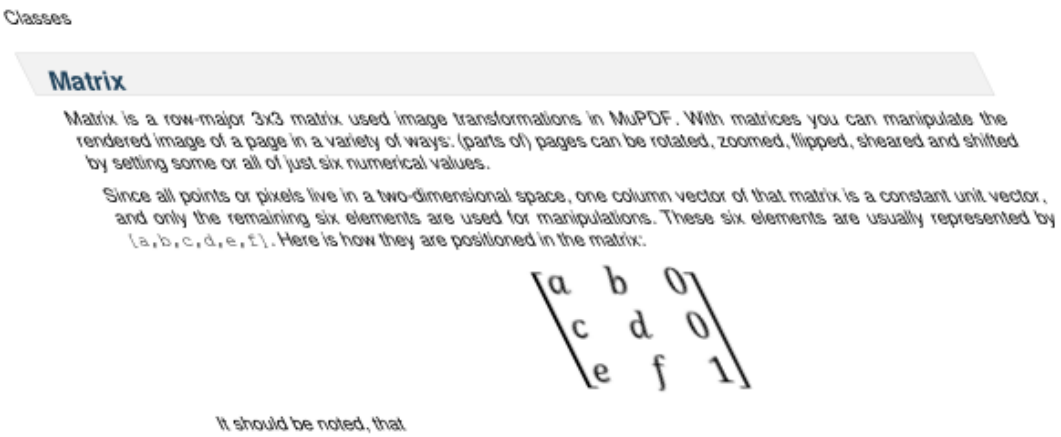
Flip up-down (`d = -1`).



5.9.4 Shearing

First a shear in Y direction (`b = 0.5`).

Second a shear in X direction (`c = 0.5`).


Classes

Matrix

Matrix is a row-major 3x3 matrix used image transformations in MuPDF. With matrices you can manipulate the rendered image of a page in a variety of ways: (parts of) pages can be rotated, zoomed, flipped, sheared and shifted by setting some or all of just six numerical values.

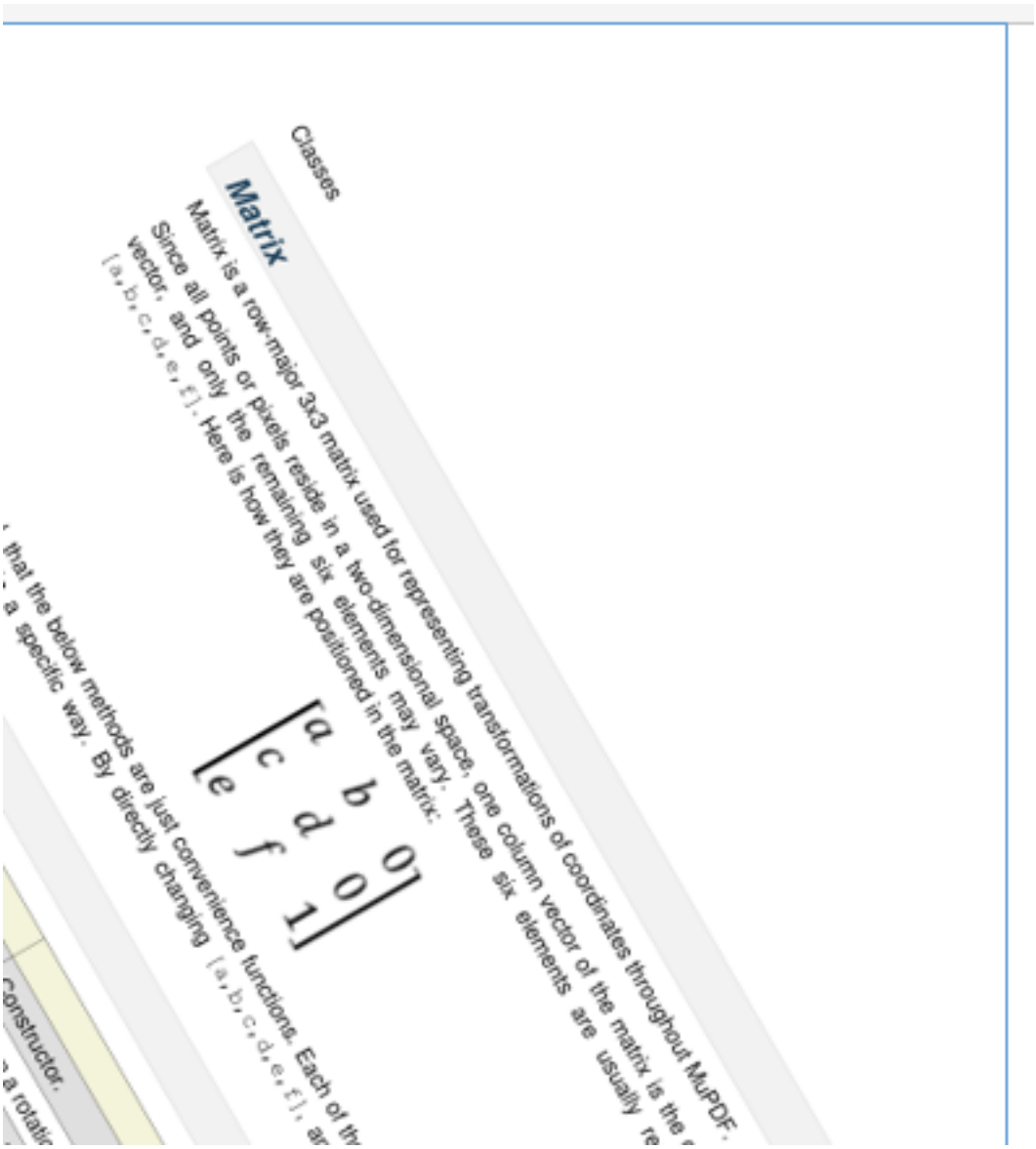
Since all points or pixels live in a two-dimensional space, one column vector of that matrix is a constant unit vector, and only the remaining six elements are used for manipulations. These six elements are usually represented by `(a,b,c,d,e,f)`. Here is how they are positioned in the matrix:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

It should be noted, that

5.9.5 Rotating

Finally a rotation by 30 clockwise degrees (`preRotate(-30)`).



5.10 Outline

`outline` (or “bookmark”), is a property of `Document`. If not `None`, it stands for the first outline item of the document. Its properties in turn define the characteristics of this item and also point to other outline items in “horizontal” or downward direction. The full tree of all outline items for e.g. a conventional table of contents (TOC) can be recovered by following these “pointers”.

Method / Attribute	Short Description
Outline.down	next item downwards
Outline.next	next item same level
Outline.page	page number (0-based)
Outline.title	title
Outline.uri	string further specifying the outline target
Outline.isExternal	target is outside this document
Outline.is_open	whether sub-outlines are open or collapsed
Outline.isOpen	whether sub-outlines are open or collapsed
Outline.dest	points to link destination details

Class API

class Outline

down
The next outline item on the next level down. Is None if the item has no kids.

Type *Outline*

next
The next outline item at the same level as this item. Is None if this is the last one in its level.

Type *Outline*

page
The page number (0-based) this bookmark points to.

Type int

title
The item's title as a string or None.

Type str

is_open
Or isOpen – an indicator showing whether any sub-outlines should be expanded (True) or be collapsed (False). This information should be interpreted by PDF display software accordingly.

Type bool

isExternal
A bool specifying whether the target is outside (True) of the current document.

Type bool

uri
A string specifying the link target. The meaning of this property should be evaluated in conjunction with isExternal. The value may be None, in which case isExternal == False. If uri starts with file://, mailto:, or an internet resource name, isExternal is True. In all other cases isExternal == False and uri points to an internal location. In case of PDF documents, this should either be #nnnn to indicate a 1-based (!) page number nnnn, or a named location. The format varies for other document types, e.g. uri = '../FixedDoc.fdoc#PG_21_LNK_84' for page number 21 (1-based) in an XPS document.

Type str

dest
The link destination details object.

Type *linkDest*

5.11 Page

Class representing a document page. A page object is created by *Document.loadPage()* or, equivalently, via indexing the document like `doc[n]` - it has no independent constructor.

There is a parent-child relationship between a document and its pages. If the document is closed or deleted, all page objects (and their respective children, too) in existence will become unusable ("orphaned"): If a page property or method is being used, an exception is raised.

Several page methods have a *Document* counterpart for convenience. At the end of this chapter you will find a synopsis.

5.11.1 Adding Page Content

This is available for PDF documents only. There are basically two groups of methods:

1. **Methods making permanent changes.** This group contains `insertText()`, `insertTextbox()` and all `draw*()` methods. They provide "stand-alone", shortcut versions for the same-named methods of the *Shape* class. For detailed descriptions have a look in that chapter. Some remarks on the relationship between the *Page* and *Shape* methods:

- In contrast to *Shape*, the results of page methods are not interconnected: they do not share properties like colors, line width / dashing, morphing, etc.
- Each page `draw*()` method invokes a *Shape.finish()* and then a *Shape.commit()* and consequently accepts the combined arguments of both these methods.
- Text insertion methods (`insertText()` and `insertTextbox()`) do not need *Shape.finish()* and therefore only invoke *Shape.commit()*.

2. **Methods adding annotations.** Annotations can be added, modified and deleted without necessarily having full document permissions. Their effect is **not permanent** in the sense, that manipulating them does not require to rebuild the document. **Adding** and **deleting** annotations are page methods. **Changing** existing annotations is possible via methods of the *Annot* class.

Method / Attribute	Short Description
Page.addCaretAnnot()	PDF only: add a caret annotation
Page.addCircleAnnot()	PDF only: add a circle annotation
Page.addFileAnnot()	PDF only: add a file attachment annotation
Page.addFreetextAnnot()	PDF only: add a text annotation
Page.addHighlightAnnot()	PDF only: add a "highlight" annotation
Page.addInkAnnot()	PDF only: add an ink annotation
Page.addLineAnnot()	PDF only: add a line annotation
Page.addPolygonAnnot()	PDF only: add a polygon annotation
Page.addPolylineAnnot()	PDF only: add a multi-line annotation
Page.addRectAnnot()	PDF only: add a rectangle annotation
Page.addSquigglyAnnot()	PDF only: add a "squiggly" annotation
Page.addStampAnnot()	PDF only: add a "rubber stamp" annotation
Page.addStrikeoutAnnot()	PDF only: add a "strike-out" annotation
Page.addTextAnnot()	PDF only: add a comment
Page.addUnderlineAnnot()	PDF only: add an "underline" annotation

Continued on next page

Table 2 – continued from previous page

Method / Attribute	Short Description
Page.addWidget()	PDF only: add a PDF Form field
Page.annots()	return a generator over the annots on the page
Page.bound()	rectangle of the page
Page.deleteAnnot()	PDF only: delete an annotation
Page.deleteLink()	PDF only: delete a link
Page.drawBezier()	PDF only: draw a cubic Bezier curve
Page.drawCircle()	PDF only: draw a circle
Page.drawCurve()	PDF only: draw a special Bezier curve
Page.drawLine()	PDF only: draw a line
Page.drawOval()	PDF only: draw an oval / ellipse
Page.drawPolyline()	PDF only: connect a point sequence
Page.drawRect()	PDF only: draw a rectangle
Page.drawSector()	PDF only: draw a circular sector
Page.drawSquiggle()	PDF only: draw a squiggly line
Page.drawZigzag()	PDF only: draw a zig-zagged line
Page.getFontList()	PDF only: get list of used fonts
Page.getImageBbox()	PDF only: get bbox of inserted image
Page.getImageList()	PDF only: get list of used images
Page.getLinks()	get all links
Page.getPixmap()	create a *Pixmap*
Page.getSVGimage()	create a page image in SVG format
Page.getText()	extract the page's text
Page.getTextPage()	create a TextPage for the page
Page.insertFont()	PDF only: insert a font for use by the page
Page.insertImage()	PDF only: insert an image
Page.insertLink()	PDF only: insert a link
Page.insertText()	PDF only: insert text
Page.insertTextbox()	PDF only: insert a text box
Page.links()	return a generator of the links on the page
Page.loadLinks()	return the first link on a page
Page.newShape()	PDF only: start a new *Shape*
Page.searchFor()	search for a string
Page.setCropBox()	PDF only: modify the visible page
Page.setRotation()	PDF only: set page rotation
Page.showPDFpage()	PDF only: display PDF page image
Page.updateLink()	PDF only: modify a link
Page.widgets()	return a generator over the fields on the page
Page.CropBox	the page's /CropBox
Page.CropBoxPosition	displacement of the /CropBox
Page.firstAnnot	first *Annot* on the page
Page.firstLink	first *Link* on the page
Page.firstWidget	first widget (form field) on the page
Page.MediaBox	the page's /MediaBox
Page.MediaBoxSize	bottom-right point of /MediaBox
Page.number	page number
Page.parent	owning document object
Page.rect	rectangle (mediabox) of the page
Page.rotation	PDF only: page rotation
Page.xref	PDF *xref*

Class API

class Page

bound()
Determine the rectangle (before transformation) of the page. Same as property *Page.rect* below. For PDF documents this **usually** also coincides with objects /MediaBox and /CropBox, but not always. The best description hence is probably "/CropBox, transformed such that top-left coordinates are (0, 0)". Also see attributes *Page.CropBox* and *Page.MediaBox*.

Return type *Rect*

addCaretAnnot(*point*)
New in version 1.16.0: PDF only: Add a caret icon. A caret annotation is a visual symbol that indicates the presence of text edits.

Parameters point (*point_like*) – the top left point of a 20 x 20 rectangle containing the MuPDF-provided icon.

Return type *Annot*

Returns the created annotation.

addTextAnnot(*point*, *text*, *icon="Note"*)
PDF only: Add a comment icon ("sticky note") with accompanying text.

Parameters

- point (*point_like*) – the top left point of a 20 x 20 rectangle containing the MuPDF-provided "note" icon.
- text (*str*) – the commentary text. This will be shown on double clicking or hovering over the icon. May contain any Latin characters.
- icon (*str*) – New in version 1.16.0: choose one of "Note" (default), "Comment", "Help", "Insert", "Key", "NewParagraph", "Paragraph" as the visual symbol for the embodied text[76].

Return type *Annot*

Returns the created annotation.

addFreetextAnnot(*rect*, *text*, *fontsize=12*, *fontname="helv"*, *text_color=0*, *fill_color=1*, *rotate=0*)
PDF only: Add text in a given rectangle.

Parameters

- rect (*rect_like*) – the rectangle into which the text should be inserted. Text is automatically wrapped to a new line at box width. Lines not fitting into the box will be invisible.
- text (*str*) – the text. May contain any Latin characters.
- fontsize (*float*) – the font size. Default is 12.

[76] You are generally free to choose any of the *Annotation Icons in MuPDF* you consider adequate.

- `fontname` (*str*) – the font name. Default is "Helv". Accepted alternatives are "Cour", "TiRo", "ZaDb" and "Symb". The name may be abbreviated to the first two characters, like "Co" for "Cour". Lower case is also accepted.
- `text_color` (*sequence*, *float*) – New in version 1.16.0: the text color. Default is black.
- `fill_color` (*sequence*, *float*) – New in version 1.16.0: the fill color. Default is white.
- `rotate` (*int*) – the text orientation. Accepted values are 0, 90, 270, invalid entries are set to zero.

Return type *Annot*

Returns the created annotation. Color properties **can only be changed** using special parameters of *Annot.update()*. There, you can also set a border color different from the text color.

`addFileAnnot`(*pos*, *buffer*, *filename*, *ufilename=None*, *desc=None*, *icon="PushPin"*)

PDF only: Add a file attachment annotation with a "PushPin" icon at the specified location.

Parameters

- `pos` (*point_like*) – the top-left point of a 18x18 rectangle containing the MuPDF-provided "PushPin" icon.
- `buffer` (*bytes*, *bytearray*, *BytesIO*) – the data to be stored (actual file content, any data, etc.).

  Changed in version 1.14.13: `io.BytesIO` is now also supported.
- `filename` (*str*) – the filename to associate with the data.
- `ufilename` (*str*) – the optional PDF unicode version of filename. Defaults to filename.
- `desc` (*str*) – an optional description of the file. Defaults to filename.
- `icon` (*str*) – New in version 1.16.0: choose one of "PushPin" (default), "Graph", "Paperclip", "Tag" as the visual symbol for the attached data[76].

Return type *Annot*

Returns the created annotation. Use methods of *Annot* to make any changes.

`addInkAnnot`(*list*)

PDF only: Add a "freehand" scribble annotation.

Parameters `list` (*sequence*) – a list of one or more lists, each containing *point_like* items. Each item in these sublists is interpreted as a *Point* through which a connecting line is drawn. Separate sublists thus represent separate drawing lines.

Return type *Annot*

Returns the created annotation in default appearance (black line of width 1). Use annotation methods with a subsequent *Annot.update()* to modify.

`addLineAnnot`(*p1*, *p2*)

PDF only: Add a line annotation.

Parameters

- `p1` (*point_like*) – the starting point of the line.
- `p2` (*point_like*) – the end point of the line.

Return type *Annot*

Returns the created annotation. It is drawn with line color black and line width 1. To change, or attach other information (like author, creation date, line properties, colors, line ends, etc.) use methods of *Annot*. The **rectangle** is automatically created to contain both points, each one surrounded by a circle of radius 3 (= 3 * line width) to make room for any line end symbols. Use methods of *Annot* to make any changes.

`addRectAnnot(`*rect*`)`

`addCircleAnnot(`*rect*`)`

PDF only: Add a rectangle, resp. circle annotation.

Parameters `rect` (*rect_like*) – the rectangle in which the circle or rectangle is drawn, must be finite and not empty. If the rectangle is not equal-sided, an ellipse is drawn.

Return type *Annot*

Returns the created annotation. It is drawn with line color black, no fill color and line width 1. Use methods of *Annot* to make any changes.

`addPolylineAnnot(`*points*`)`

`addPolygonAnnot(`*points*`)`

PDF only: Add an annotation consisting of lines which connect the given points. A **Polygon's** first and last points are automatically connected, which does not happen for a **PolyLine**. The **rectangle** is automatically created as the smallest rectangle containing the points, each one surrounded by a circle of radius 3 (= 3 * line width). The following shows a 'PolyLine' that has been modified with colors and line ends.

Parameters `points` (*list*) – a list of *point_like* objects.

Return type *Annot*

Returns the created annotation. It is drawn with line color black, no fill color and line width 1. Use methods of *Annot* to make any changes to achieve something like this:

`addUnderlineAnnot(`*quads*`)`

`addStrikeoutAnnot(`*quads*`)`

`addSquigglyAnnot(`*quads*`)`

`addHighlightAnnot(`*quads*`)`

PDF only: These annotations are normally used for marking text which has previously been located (for example via *searchFor()*). But the actual presence of text within the specified area(s) is neither checked nor required. So you are free to "mark" anything.

Standard colors are chosen per annotation type: **yellow** for highlighting, **red** for strike out, **green** for underlining, and **magenta** for wavy underlining.

The methods convert the argument into a list of *Quad* objects. The **annotation** rectangle is calculated to envelop these quadrilaterals.

Note: *searchFor()* supports *Quad* objects as an output option. Hence the following two statements are sufficient to locate and mark every occurrence of string "pymupdf" with **one common** annotation:

```
>>> quads = page.searchFor("pymupdf", hit_max=100, quads=True)
>>> page.addHighlightAnnot(quads)
```

Parameters quads (*rect_like*, *quad_like*, *list*, *tuple*) – Changed in version 1.14.20: the rectangles or quads containing the to-be-marked text (locations). A list or tuple must consist of *rect_like* or *quad_like* items (or even a mixture of either). You should prefer using quads, because this will automatically support non-horizontal text and avoid rectangle-to-quad conversion effort.

Return type *Annot*

Returns the created annotation. To change colors, set the "stroke" color accordingly (*Annot.setColors()*) and then perform an *Annot.update()*.

addStampAnnot(*rect*, *stamp=0*)

PDF only: Add a "rubber stamp" like annotation to e.g. indicate the document's intended use ("DRAFT", "CONFIDENTIAL", etc.).

Parameters

- rect (*rect_like*) – rectangle where to place the annotation.
- stamp (*int*) – id number of the stamp text. For available stamps see *Stamp Annotation Icons*.

Note: The stamp's text (e.g. "APPROVED") and its border line will automatically be sized and put centered in the given rectangle. *Annot.rect* is automatically calculated to fit and will usually be smaller than this parameter. The appearance can be changed using *Annot.setOpacity()* and by setting the "stroke" color (no "fill" color supported).

addWidget(*widget*)

PDF only: Add a PDF Form field ("widget") to a page. This also **turns the PDF into a Form PDF**. Because of the large amount of different options available for widgets, we have developed a new

class *Widget*, which contains the possible PDF field attributes. It must be used for both, form field creation and updates.

Parameters `widget` (*Widget*) – a *Widget* object which must have been created upfront.

Returns a widget annotation.

`deleteAnnot`(*annot*)
PDF only: Delete the specified annotation from the page and return the next one.

Changed in version 1.16.6: The removal will now include any bound 'Popup' or response annotations and related objects.

Parameters `annot` (*Annot*) – the annotation to be deleted.

Return type *Annot*

Returns the annotation following the deleted one. Please remember that physical removal will take place only with saving to a new file with a positive garbage collection option.

`deleteLink`(*linkdict*)
PDF only: Delete the specified link from the page. The parameter must be an **original item** of *getLinks()* (see below). The reason for this is the dictionary's `"xref"` key, which identifies the PDF object to be deleted.

Parameters `linkdict` (*dict*) – the link to be deleted.

`insertLink`(*linkdict*)
PDF only: Insert a new link on this page. The parameter must be a dictionary of format as provided by *getLinks()* (see below).

Parameters `linkdict` (*dict*) – the link to be inserted.

`updateLink`(*linkdict*)
PDF only: Modify the specified link. The parameter must be a (modified) **original item** of *getLinks()* (see below). The reason for this is the dictionary's `"xref"` key, which identifies the PDF object to be changed.

Parameters `linkdict` (*dict*) – the link to be modified.

`getLinks`()
Retrieves **all** links of a page.

Return type list

Returns A list of dictionaries. For a description of the dictionary entries see below. Always use this or the *Page.links()* method if you intend to make changes to the links of a page.

`links`(*kinds=None*)
New in version 1.16.4: Return a generator over the page's links. The results equal the entries of *Page.getLinks()*.

Parameters `kinds` (*sequence*) – a sequence of integers to down-select to one or more link kinds. Default is all links. Example: `kinds=(fitz.LINK_GOTO,)` will only return internal links.

Return type generator

Returns an entry of *Page.getLinks()* for each iteration.

`annots`(*types=None*)
New in version 1.16.4: Return a generator over the page's annotations.

Parameters types (*sequence*) – a sequence of integers to down-select to one or annotation types. Default is all annotations. Example: types=(fitz.PDF_ANNOT_FREETEXT, fitz.PDF_ANNOT_TEXT) will only return 'FreeText' and 'Text' annotations.

Return type generator

Returns an *Annot* for each iteration.

widgets(*types=None*)

New in version 1.16.4: Return a generator over the page's form fields.

Parameters types (*sequence*) – a sequence of integers to down-select to one or more widget types. Default is all form fields. Example: types=(fitz.PDF_WIDGET_TYPE_TEXT,) will only return 'Text' fields.

Return type generator

Returns a *Widget* for each iteration.

insertText(*point*, *text*, *fontsize=11*, *fontname="helv"*, *fontfile=None*, *idx=0*, *color=None*, *fill=None*, *render_mode=0*, *border_width=1*, *encoding=TEXT_ENCODING_LATIN*, *rotate=0*, *morph=None*, *overlay=True*)

PDF only: Insert text starting at *point_like* point. See *Shape.insertText()*.

insertTextbox(*rect*, *buffer*, *fontsize=11*, *fontname="helv"*, *fontfile=None*, *idx=0*, *color=None*, *fill=None*, *render_mode=0*, *border_width=1*, *encoding=TEXT_ENCODING_LATIN*, *expandtabs=8*, *align=TEXT_ALIGN_LEFT*, *charwidths=None*, *rotate=0*, *morph=None*, *overlay=True*)

PDF only: Insert text into the specified *rect_like* rect. See *Shape.insertTextbox()*.

drawLine(*p1*, *p2*, *color=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*)

PDF only: Draw a line from p1 to p2 (*point_like* s). See *Shape.drawLine()*.

drawZigzag(*p1*, *p2*, *breadth=2*, *color=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*)

PDF only: Draw a zigzag line from p1 to p2 (*point_like* s). See *Shape.drawZigzag()*.

drawSquiggle(*p1*, *p2*, *breadth=2*, *color=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*)

PDF only: Draw a squiggly (wavy, undulated) line from p1 to p2 (*point_like* s). See *Shape.drawSquiggle()*.

drawCircle(*center*, *radius*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*)

PDF only: Draw a circle around center (*point_like*) with a radius of radius. See *Shape.drawCircle()*.

drawOval(*quad*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*)

PDF only: Draw an oval (ellipse) within the given *rect_like* or *quad_like*. See *Shape.drawOval()*.

drawSector(*center*, *point*, *angle*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *fullSector=True*, *overlay=True*, *closePath=False*, *morph=None*)

PDF only: Draw a circular sector, optionally connecting the arc to the circle's center (like a piece of pie). See *Shape.drawSector()*.

drawPolyline(*points*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *closePath=False*, *morph=None*)

PDF only: Draw several connected lines defined by a sequence of *point_like* s. See *Shape.drawPolyline()*.

drawBezier(*p1*, *p2*, *p3*, *p4*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *closePath=False*, *morph=None*)
PDF only: Draw a cubic BÃ©zier curve from `p1` to `p4` with the control points `p2` and `p3` (all are :data‘point_like‘ s). See *Shape.drawBezier()*.

drawCurve(*p1*, *p2*, *p3*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *closePath=False*, *morph=None*)
PDF only: This is a special case of `drawBezier()`. See *Shape.drawCurve()*.

drawRect(*rect*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*)
PDF only: Draw a rectangle. See *Shape.drawRect()*.

Note: An efficient way to background-color a PDF page with the old Python paper color is

```
>>> col = fitz.utils.getColor("py_color")
>>> page.drawRect(page.rect, color=col, fill=col, overlay=False)
```

---

insertFont(*fontname="helv"*, *fontfile=None*, *fontbuffer=None*, *set_simple=False*, *encoding=TEXT_ENCODING_LATIN*)
PDF only: Add a new font to be used by text output methods and return its *xref*. If not already present in the file, the font definition will be added. Supported are the built-in *Base14_Fonts* and the CJK fonts via **“reserved”** fontnames. Fonts can also be provided as a file path or a memory area containing the image of a font file.

Parameters `fontname` (*str*) – The name by which this font shall be referenced when outputting text on this page. In general, you have a “free” choice here (but consult the *Adobe PDF Reference 1.7*, page 56, section 3.2.4 for a formal description of building legal PDF names). However, if it matches one of the *Base14_Fonts* or one of the CJK fonts, `fontfile` and `fontbuffer` **are ignored**.

In other words, you cannot insert a font via `fontfile` / `fontbuffer` and also give it a reserved `fontname`.

Note: A reserved fontname can be specified in any mixture of upper or lower case and still match the right built-in font definition: fontnames “helv”, “Helv”, “HELV”, “Helvetica”, etc. all lead to the same font definition “Helvetica”. But from a *Page* perspective, these are **different references**. You can exploit this fact when using different `encoding` variants (Latin, Greek, Cyrillic) of the same font on a page.

Parameters

- `fontfile` (*str*) – a path to a font file. If used, `fontname` must be **different from all reserved names**.
- `fontbuffer` (*bytes/bytearray*) – the memory image of a font file. If used, `fontname` must be **different from all reserved names**. This parameter would typically be used to transfer fonts between different pages of the same or different PDFs.
- `set_simple` (*int*) – applicable for `fontfile` / `fontbuffer` cases only: enforce treatment as a “simple” font, i.e. one that only uses character codes up to 255.
- `encoding` (*int*) – applicable for the “Helvetica”, “Courier” and “Times” sets of *Base14_Fonts* only. Select one of the available encodings Latin (0), Cyrillic (2) or Greek (1). Only use the default (0 = Latin) for “Symbol” and “ZapfDingBats”.

Rytpe int

Returns the *xref* of the installed font.

Note: Built-in fonts will not lead to the inclusion of a font file. So the resulting PDF file will remain small. However, your PDF viewer software is responsible for generating an appropriate appearance – and there **exist** differences on whether or how each one of them does this. This is especially true for the CJK fonts. But also Symbol and ZapfDingbats are incorrectly handled in some cases. Following are the **Font Names** and their correspondingly installed **Base Font** names:

Base-14 Fonts[73]

Font Name	Installed Base Font	Comments
helv	Helvetica	normal
heit	Helvetica-Oblique	italic
hebo	Helvetica-Bold	bold
hebi	Helvetica-BoldOblique	bold-italic
cour	Courier	normal
coit	Courier-Oblique	italic
cobo	Courier-Bold	bold
cobi	Courier-BoldOblique	bold-italic
tiro	Times-Roman	normal
tiit	Times-Italic	italic
tibo	Times-Bold	bold
tibi	Times-BoldItalic	bold-italic
symb	Symbol	[75]
zadb	ZapfDingbats	[75]

CJK Fonts[74] (China, Japan, Korea)

Font Name	Installed Base Font	Comments
china-s	Heiti	simplified Chinese
china-ss	Song	simplified Chinese (serif)
china-t	Fangti	traditional Chinese
china-ts	Ming	traditional Chinese (serif)
japan	Gothic	Japanese
japan-s	Mincho	Japanese (serif)
korea	Dotum	Korean
korea-s	Batang	Korean (serif)

`insertImage`(*rect*, *filename=None*, *pixmap=None*, *stream=None*, *rotate=0*, *keep_proportion=True*, *overlay=True*)

PDF only: Put an image inside the given rectangle. The image can be taken from a pixmap, a file or a memory area - of these parameters **exactly one** must be specified.

[73] If your existing code already uses the installed base name as a font reference (as it was supported by PyMuPDF versions earlier than 1.14), this will continue to work.

[75] Not all PDF readers display these fonts at all. Some others do, but use a wrong character spacing, etc.

[74] Not all PDF reader software (including internet browsers and office software) display all of these fonts. And if they do, the difference between the **serifed** and the **non-serifed** version may hardly be noticable. But serifed and non-serifed versions lead to different installed base fonts, thus providing an option to be displayable with your specific PDF viewer.

Changed in version 1.14.11: By default, the image keeps its aspect ratio.

Parameters

- `rect` (*rect_like*) – where to put the image on the page. Only the rectangle part which is inside the page is used. This intersection must be finite and not empty.

  Changed in version 1.14.13: The image is now always placed **centered** in the rectangle, i.e. the center of the image and the rectangle coincide.

- `filename` (*str*) – name of an image file (all formats supported by MuPDF – see *Supported Input Image Formats*). If the same image is to be inserted multiple times, choose one of the other two options to avoid some overhead.

- `stream` (*bytes , bytearray , io.BytesIO*) – image in memory (all formats supported by MuPDF – see *Supported Input Image Formats*). This is the most efficient option.

  Changed in version 1.14.13: `io.BytesIO` is now also supported.

- `pixmap` (*Pixmap*) – a pixmap containing the image.

- `rotate` (*int*) – New in version v1.14.11: rotate the image. Must be an integer multiple of 90 degrees. If you need a rotation by an arbitrary angle, consider converting the image to a PDF (*Document.convertToPDF()*) first and then use *Page.showPDFpage()* instead.

- `keep_proportion` (*bool*) – New in version v1.14.11: maintain the aspect ratio of the image.

For a description of `overlay` see *Common Parameters*.

This example puts the same image on every page of a document:

```
>>> doc = fitz.open(...)
>>> rect = fitz.Rect(0, 0, 50, 50)        # put thumbnail in upper left corner
>>> img = open("some.jpg", "rb").read()   # an image file
>>> for page in doc:
        page.insertImage(rect, stream = img)
>>> doc.save(...)
```

Note:

1. If that same image had already been present in the PDF, then only a reference to it will be inserted. This of course considerably saves disk space and processing time. But to detect this fact, existing PDF images need to be compared with the new one. This is achieved by storing an MD5 code for each image in a table and only compare the new image's MD5 code against the table entries. Generating this MD5 table, however, is done when the first image is inserted - which therefore may have an extended response time.

2. You can use this method to provide a background or foreground image for the page, like a copyright, a watermark. Please remember, that watermarks require a transparent image ...

3. The image may be inserted uncompressed, e.g. if a `Pixmap` is used or if the image has an alpha channel. Therefore, consider using `deflate=True` when saving the file.

4. The image is stored in the PDF in its original quality. This may be much better than you ever need for your display. In this case consider decreasing the image size before inserting it – e.g. by using the pixmap option and then shrinking it or scaling it down (see *Pixmap* chapter). The file size savings can be very significant.

5. The most efficient way to display the same image on multiple pages is another method: *showPDFpage()*. Consult *Document.convertToPDF()* for how to obtain intermediary PDFs usable for that method. Demo script fitz-logo.py[69] implements a fairly complete approach.

---

getText(*opt="text"*, *flags=None*)

Retrieves the content of a page in a variety of formats. This is a wrapper for *TextPage* methods by choosing the output option as follows:

- "text" – *TextPage.extractTEXT()*, default
- "blocks" – *TextPage.extractBLOCKS()*
- "words" – *TextPage.extractWORDS()*
- "html" – *TextPage.extractHTML()*
- "xhtml" – *TextPage.extractXHTML()*
- "xml" – *TextPage.extractXML()*
- "dict" – *TextPage.extractDICT()*
- "json" – *TextPage.extractJSON()*
- "rawdict" – *TextPage.extractRAWDICT()*

Parameters

- opt (*str*) – A string indicating the requested format, one of the above. A mixture of upper and lower case is supported.

  Changed in version 1.16.3: Values "words" and "blocks" are now also accepted.

- flags (*int*) – New in version 1.16.2: indicator bits to control whether to include images or how text should be handled with respect to (white) spaces and ligatures. See *Preserve Text Flags* for available indicators and *Text Extraction Flags Defaults* for default settings.

Return type *str, list, dict*

Returns The page's content as a string, list or as a dictionary. Refer to the corresponding *TextPage* method for details.

Note: You can use this method as a **document conversion tool** from any supported document type (not only PDF!) to one of TEXT, HTML, XHTML or XML documents.

---

getTextPage(*flags=3*)

New in version 1.16.5: Create a *TextPage* for the page. This method avoids using an intermediate *DisplayList*.

Parameters flags (*in*) – indicator bits controlling the content available for subsequent extraction – see the parameter of *Page.getText()*.

Returns *TextPage*

getFontList(*full=False*)

PDF only: Return a list of fonts referenced by the page. Wrapper for *Document.getPageFontList()*.

[69] https://github.com/pymupdf/PyMuPDF/blob/master/demo/fitz-logo.py

getImageList(*full=False*)

PDF only: Return a list of images referenced by the page. Wrapper for *Document.getPageImageList()*.

getImageBbox(*item*)

Parameters item (*list*) – an item of the list *Page.getImageList()* with full=True specified.

Return type *Rect*

Returns the boundary box of the image. .. versionchanged:: 1.16.7 If the page in fact does not display this image, an infinite rectangle is returned now. In previous versions, an exception was raised.

Warning: The method internally cleans the page's /Contents object(s) using *Page._cleanContents()*. Please consult its description for implications.

Note:

- Be aware that *Page.getImageList()* may contain "dead" entries, i.e. there may be image references which – although present in the PDF – are **not displayed** by this page. In this case an exception is raised.
- This function is still somewhat **experimental**: it does not yet cover all possibilities of how an image location might have been coded, but instead makes some simplifying assumptions. As a result you occasionally may find the bbox incorrectly calculated. In contrast, image blocks returned by *Page.getText()* ("dict" or "rawdict" options) do contain a correct bbox on the one hand, but on the other hand do **not allow an (easy) identification** of the image as a PDF object. There are however ways to match these information pieces – please consult the recipes chapter.

getSVGimage(*matrix=fitz.Identity*)

Create an SVG image from the page. Only full page images are currently supported.

Parameters matrix (*matrix_like*) – a matrix, default is *Identity*.

Returns a UTF-8 encoded string that contains the image. Because SVG has XML syntax it can be saved in a text file with extension .svg.

getPixmap(*matrix=fitz.Identity*, *colorspace=fitz.csRGB*, *clip=None*, *alpha=False*, *annots=True*)

Create a pixmap from the page. This is probably the most often used method to create a pixmap.

Parameters

- matrix (*matrix_like*) – default is *Identity*.
- colorspace (str or *Colorspace*) – Defines the required colorspace, one of "GRAY", "RGB" or "CMYK" (case insensitive). Or specify a *Colorspace*, ie. one of the predefined ones: *csGRAY*, *csRGB* or *csCMYK*.
- clip (*irect_like*) – restrict rendering to this area.
- alpha (*bool*) – whether to add an alpha channel. Always accept the default False if you do not really need transparency. This will save a lot of memory (25% in case of RGB ... and pixmaps are typically **large**!), and also processing time. Also

note an **important difference** in how the image will be rendered: with `True` the pixmap's samples area will be pre-cleared with `0x00`. This results in **transparent** areas where the page is empty. With `False` the pixmap's samples will be pre-cleared with `0xff`. This results in **white** where the page has nothing to show.

Changed in version 1.14.17: The default alpha value is now `False`.

– Generated with `alpha=True`

– Generated with `alpha=False`

- `annots` (*bool*) – New in version 1.16.0: whether to also render any annotations on the page. You can create pixmaps for each annotation separately.

Return type *Pixmap*

Returns Pixmap of the page.

`loadLinks()`
Return the first link on a page. Synonym of property *firstLink*.

Return type *Link*

Returns first link on the page (or `None`).

`setRotation(`*rotate*`)`
PDF only: Sets the rotation of the page.

Parameters `rotate` (*int*) – An integer specifying the required rotation in degrees. Must be an integer multiple of 90.

`showPDFpage(`*rect*, *docsrc*, *pno=0*, *keep_proportion=True*, *overlay=True*, *rotate=0*, *clip=None*`)`
PDF only: Display a page of another PDF as a **vector image** (otherwise similar to *Page.insertImage()*). This is a multi-purpose method. For example, you can use it to

- create "n-up" versions of existing PDF files, combining several input pages into **one output page** (see example 4-up.py[70]),
- create "posterized" PDF files, i.e. every input page is split up in parts which each create a separate output page (see posterize.py[71]),
- include PDF-based vector images like company logos, watermarks, etc., see svg-logo.py[72], which puts an SVG-based logo on each page (requires additional packages to deal with SVG-to-PDF conversions).

Changed in version 1.14.11: Parameter `reuse_xref` has been deprecated.

Parameters

- `rect` (*rect_like*) – where to place the image on current page. Must be finite and its intersection with the page must not be empty.

  Changed in version 1.14.11: Position the source rectangle centered in this rectangle.
- `docsrc` (*Document*) – source PDF document containing the page. Must be a different document object, but may be the same file.
- `pno` (*int*) – page number (0-based, in $range(-\infty, docsrc.pageCount)$) to be shown.
- `keep_proportion` (*bool*) – whether to maintain the width-height-ratio (default). If false, all 4 corners are always positioned on the border of the target rectangle – whatever the rotation value. In general, this will deliver distorted and /or non-rectangular images.
- `overlay` (*bool*) – put image in foreground (default) or background.
- `rotate` (*float*) – New in version 1.14.10: show the source rectangle rotated by some angle.

  Changed in version 1.14.11: Any angle is now supported.
- `clip` (*rect_like*) – choose which part of the source page to show. Default is the full page, else must be finite and its intersection with the source page must not be empty.

Note: In contrast to method *Document.insertPDF()*, this method does not copy annotations or links, so they are not shown. But all its **other resources (text, images, fonts, etc.)** will be imported into the current PDF. They will therefore appear in text extractions and in *getFontList()* and *getImageList()* lists – even if they are not contained in the visible area given by `clip`.

Example: Show the same source page, rotated by 90 and by -90 degrees:

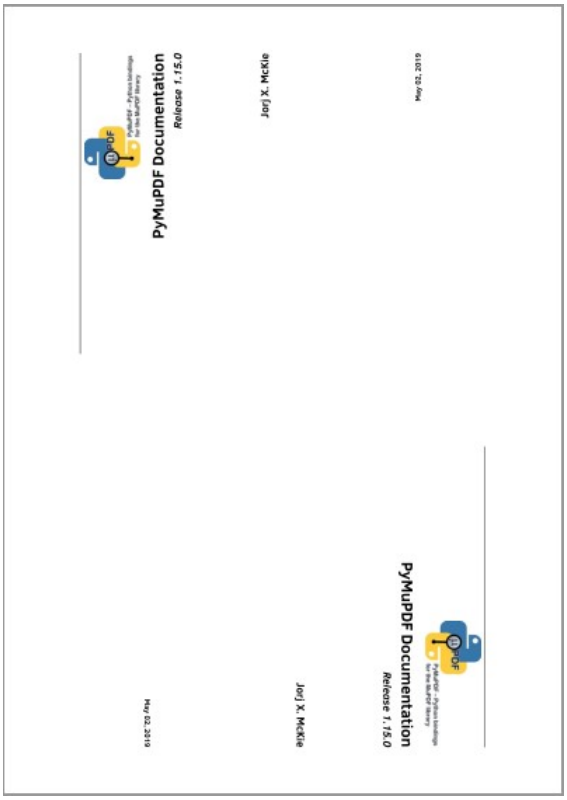
```
>>> doc = fitz.open()  # new empty PDF
>>> page=doc.newPage()  # new page in A4 format
>>>
>>> # upper half page
>>> r1 = fitz.Rect(0, 0, page.rect.width, page.rect.height/2)
>>>
>>> # lower half page
```

(continues on next page)

[70] https://github.com/pymupdf/PyMuPDF/blob/master/examples/4-up.py
[71] https://github.com/pymupdf/PyMuPDF/blob/master/examples/posterize.py
[72] https://github.com/pymupdf/PyMuPDF/blob/master/examples/svg-logo.py

(continued from previous page)

```
>>> r2 = r1 + (0, page.rect.height/2, 0, page.rect.height/2)
>>>
>>> src = fitz.open("PyMuPDF.pdf")  # show page 0 of this
>>>
>>> page.showPDFpage(r1, src, 0, rotate=90)
>>> page.showPDFpage(r2, src, 0, rotate=-90)
>>> doc.save("show.pdf")
```



newShape()
PDF only: Create a new *Shape* object for the page.

Return type *Shape*

Returns a new *Shape* to use for compound drawings. See description there.

searchFor(*text*, *hit_max=16*, *quads=False*, *flags=None*)
Searches for `text` on a page. Wrapper for *TextPage.search()*.

Parameters

- `text` (*str*) – Text to search for. Upper / lower case is ignored. The string may contain spaces.
- `hit_max` (*int*) – Maximum number of occurrences accepted.
- `quads` (*bool*) – Return *Quad* instead of *Rect* objects.
- `flags` (*int*) – Control the data extracted by the underlying *TextPage*. Default is 0 (ligatures are dissolved, white space is replaced with space and excessive spaces are not suppressed).

Return type list

Returns

A list of *Rect* s (resp. *Quad* s) each of which – **normally!** – surrounds one occurrence of `text`. **However:** if the search string spreads across more than one line, then a separate item is recorded in the list for each part of the string per line. So, if you are looking for "search string" and the two words happen to be located on separate lines, two entries will be recorded in the list: one for "search" and one for "string".

Note: In this way, the effect supports multi-line text marker annotations.

`setCropBox(r)`

PDF only: change the visible part of the page.

Parameters `r` (*rect_like*) – the new visible area of the page.

After execution, *Page.rect* will equal this rectangle, shifted to the top-left position (0, 0). Example session:

```
>>> page = doc.newPage()
>>> page.rect
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> page.CropBox                    # CropBox and MediaBox still equal
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> # now set CropBox to a part of the page
>>> page.setCropBox(fitz.Rect(100, 100, 400, 400))
>>> # this will also change the "rect" property:
>>> page.rect
fitz.Rect(0.0, 0.0, 300.0, 300.0)
>>>
>>> # but MediaBox remains unaffected
>>> page.MediaBox
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> # revert everything we did
>>> page.setCropBox(page.MediaBox)
>>> page.rect
fitz.Rect(0.0, 0.0, 595.0, 842.0)
```

`rotation`

PDF only: contains the rotation of the page in degrees and `-1` for other document types.

Type int

`CropBoxPosition`

Contains the displacement of the page's `/CropBox` for a PDF, otherwise the top-left coordinates of *Page.rect*.

Type *Point*

`CropBox`

The page's `/CropBox` for a PDF, else *Page.rect*.

Type *Rect*

`MediaBoxSize`

Contains the width and height of the page's `/MediaBox` for a PDF, otherwise the bottom-right coordinates of *Page.rect*.

Type *Point*

`MediaBox`
The page's `/MediaBox` for a PDF, otherwise *`Page.rect`*.

Type *Rect*

Note: For most PDF documents and for all other types, `page.rect == page.CropBox == page.MediaBox` is true. However, for some PDFs the visible page is a true subset of `/MediaBox`. In this case the above attributes help to correctly locate page elements.

`firstLink`
Contains the first *Link* of a page (or `None`).

Type *Link*

`firstAnnot`
Contains the first *Annot* of a page (or `None`).

Type *Annot*

`firstWidget`
Contains the first *Widget* of a page (or `None`).

Type *Widget*

`number`
The page number.

Type int

`parent`
The owning document object.

Type *Document*

`rect`
Contains the rectangle of the page. Same as result of *`Page.bound()`*.

Type *Rect*

`xref`
The page's PDF *`xref`*. Zero if not a PDF.

Type *Rect*

5.11.2 Description of `getLinks()` Entries

Each entry of the `getLinks()` list is a dictionay with the following keys:

- `kind`: (required) an integer indicating the kind of link. This is one of `LINK_NONE`, `LINK_GOTO`, `LINK_GOTOR`, `LINK_LAUNCH`, or `LINK_URI`. For values and meaning of these names refer to *Link Destination Kinds*.
- `from`: (required) a *Rect* describing the "hot spot" location on the page's visible representation (where the cursor changes to a hand image, usually).
- `page`: a 0-based integer indicating the destination page. Required for `LINK_GOTO` and `LINK_GOTOR`, else ignored.

- `to`: either a `fitz.Point`, specifying the destination location on the provided page, default is `fitz.Point(0, 0)`, or a symbolic (indirect) name. If an indirect name is specified, `page = -1` is required and the name must be defined in the PDF in order for this to work. Required for `LINK_GOTO` and `LINK_GOTOR`, else ignored.
- `file`: a string specifying the destination file. Required for `LINK_GOTOR` and `LINK_LAUNCH`, else ignored.
- `uri`: a string specifying the destination internet resource. Required for `LINK_URI`, else ignored.
- `xref`: an integer specifying the PDF *xref* of the link object. Do not change this entry in any way. Required for link deletion and update, otherwise ignored. For non-PDF documents, this entry contains `-1`. It is also `-1` for **all** entries in the `getLinks()` list, if **any** of the links is not supported by MuPDF - see the note below.

5.11.3 Notes on Supporting Links

MuPDF's support for links has changed in **v1.10a**. These changes affect link types *LINK_GOTO* and *LINK_GOTOR*.

5.11.3.1 Reading (pertains to method `getLinks()` and the `firstLink` property chain)

If MuPDF detects a link to another file, it will supply either a `LINK_GOTOR` or a `LINK_LAUNCH` link kind. In case of `LINK_GOTOR` destination details may either be given as page number (eventually including position information), or as an indirect destination.

If an indirect destination is given, then this is indicated by `page = -1`, and `link.dest.dest` will contain this name. The dictionaries in the `getLinks()` list will contain this information as the `to` value.

Internal links are always of kind `LINK_GOTO`. If an internal link specifies an indirect destination, it **will always be resolved** and the resulting direct destination will be returned. Names are **never returned for internal links**, and undefined destinations will cause the link to be ignored.

5.11.3.2 Writing

PyMuPDF writes (updates, inserts) links by constructing and writing the appropriate PDF object **source**. This makes it possible to specify indirect destinations for `LINK_GOTOR` **and** `LINK_GOTO` link kinds (pre `PDF 1.2` file formats are **not supported**).

> **Warning:** If a `LINK_GOTO` indirect destination specifies an undefined name, this link can later on not be found / read again with MuPDF / PyMuPDF. Other readers however **will** detect it, but flag it as erroneous.

Indirect `LINK_GOTOR` destinations can in general of course not be checked for validity and are therefore **always accepted**.

5.11.4 Homologous Methods of Document and Page

This is an overview of homologous methods on the *Document* and on the *Page* level.

Document Level	Page Level
`Document.getPageFontlist(pno)`	*`Page.getFontList()`*
`Document.getPageImageList(pno)`	*`Page.getImageList()`*
`Document.getPagePixmap(pno, ...)`	*`Page.getPixmap()`*
`Document.getPageText(pno, ...)`	*`Page.getText()`*
`Document.searchPageFor(pno, ...)`	*`Page.searchFor()`*

The page number `pno` is a 0-based integer $-\infty < pno < pageCount$.

Note: Most document methods (left column) exist for convenience reasons, and are just wrappers for: `Document[pno].<page method>`. So they **load and discard the page** on each execution.

However, the first two methods work differently. They only need a page's object definition statement - the page itself will **not** be loaded. So e.g. *`Page.getFontList()`* is a wrapper the other way round and defined as follows: `page.getFontList == page.parent.getPageFontList(page.number)`.

5.12 Pixmap

Pixmaps ("pixel maps") are objects at the heart of MuPDF's rendering capabilities. They represent plane rectangular sets of pixels. Each pixel is described by a number of bytes ("components") defining its color, plus an optional alpha byte defining its transparency.

In PyMuPDF, there exist several ways to create a pixmap. Except the first one, all of them are available as overloaded constructors. A pixmap can be created ...

1. from a document page (method *`Page.getPixmap()`*)
2. empty, based on *Colorspace* and *IRect* information
3. from a file
4. from an in-memory image
5. from a memory area of plain pixels
6. from an image inside a PDF document
7. as a copy of another pixmap

Note: A number of image formats is supported as input for points 3. and 4. above. See section *Supported Input Image Formats*.

Have a look at the *Collection of Recipes* section to see some pixmap usage "at work".

Method / Attribute	Short Description
Pixmap.clearWith()	clear parts of a pixmap
Pixmap.copyPixmap()	copy parts of another pixmap
Pixmap.gammaWith()	apply a gamma factor to the pixmap
Pixmap.getImageData()	return a memory area in a variety of formats
Pixmap.getPNGData()	return a PNG as a memory area
Pixmap.invertIRect()	invert the pixels of a given area
Pixmap.pixel()	return the value of a pixel
Pixmap.setPixel()	set the color of a pixel
Pixmap.setRect()	set the color of a rectangle
Pixmap.setAlpha()	set alpha values
Pixmap.shrink()	reduce size keeping proportions
Pixmap.tintWith()	tint a pixmap with a color
Pixmap.writeImage()	save a pixmap in a variety of formats
Pixmap.writePNG()	save a pixmap as a PNG file
Pixmap.alpha	transparency indicator
Pixmap.colorspace	pixmap's *Colorspace*
Pixmap.height	pixmap height
Pixmap.interpolate	interpolation method indicator
Pixmap.irect	*IRect* of the pixmap
Pixmap.n	bytes per pixel
Pixmap.samples	pixel area
Pixmap.size	pixmap's total length
Pixmap.stride	size of one image row
Pixmap.width	pixmap width
Pixmap.x	X-coordinate of top-left corner
Pixmap.xres	resolution in X-direction
Pixmap.y	Y-coordinate of top-left corner
Pixmap.yres	resolution in Y-direction

Class API

`class Pixmap`

`__init__`(*self*, *colorspace*, *irect*, *alpha*)

New empty pixmap: Create an empty pixmap of size and origin given by the rectangle. So, `irect.top_left` designates the top left corner of the pixmap, and its width and height are `irect.width` resp. `irect.height`. Note that the image area is **not initialized** and will contain crap data – use eg. *clearWith()* or *setRect()* to be sure.

Parameters

- `colorspace` (*Colorspace*) – colorspace.
- `irect` (*irect_like*) – Tte pixmap's position and dimension.
- `alpha` (*bool*) – Specifies whether transparency bytes should be included. Default is `False`.

`__init__`(*self*, *colorspace*, *source*)

Copy and set colorspace: Copy `source` pixmap converting colorspace. Any colorspace combination is possible, but source colorspace must not be `None`.

Parameters

- `colorspace` (*Colorspace*) – desired **target** colorspace. This **may also be** `None`. In this case, a "masking" pixmap is created: its *`Pixmap.samples`* will consist of the source's alpha bytes only.
- `source` (`Pixmap`) – the source pixmap.

`__init__`(*self*, *source*, *width*, *height*[, *clip*])

Copy and scale: Copy `source` pixmap choosing new width and height values. Supports partial copying and the source colorspace may be also `None`.

Parameters

- `source` (`Pixmap`) – the source pixmap.
- `width` (*float*) – desired target width.
- `height` (*float*) – desired target height.
- `clip` (*irect_like*) – a region of the source pixmap to take the copy from.

Note: If width or height are not *de facto* integers (meaning e.g. `round(width) != width`), then pixmap will be created with `alpha = 1`.

`__init__`(*self*, *source*, *alpha = 1*)

Copy and add or drop alpha: Copy `source` and add or drop its alpha channel. Identical copy if `alpha` equals `source.alpha`. If an alpha channel is added, its values will be set to 255.

Parameters

- `source` (`Pixmap`) – source pixmap.
- `alpha` (*bool*) – whether the target will have an alpha channel, default and mandatory if source colorspace is `None`.

Note: A typical use includes separation of color and transparency bytes in separate pixmaps. Some applications require this like e.g. `wx.Bitmap.FromBufferAndAlpha()` of `wxPython`:

```
>>> # 'pix' is an RGBA pixmap
>>> pixcolors = fitz.Pixmap(pix, 0)     # extract the RGB part (drop alpha)
>>> pixalpha = fitz.Pixmap(None, pix)   # extract the alpha part
>>> bm = wx.Bitmap.FromBufferAndAlpha(pix.widht, pix.height, pixcolors.samples, pixalpha.
↪samples)
```

`__init__`(*self*, *filename*)

From a file: Create a pixmap from `filename`. All properties are inferred from the input. The origin of the resulting pixmap is `(0, 0)`.

Parameters `filename` (*str*) – Path of the image file.

`__init__`(*self*, *stream*)

From memory: Create a pixmap from a memory area. All properties are inferred from the input. The origin of the resulting pixmap is `(0, 0)`.

Parameters `stream` (*bytes*, *bytearray*, *BytesIO*) – Data containing a complete, valid image. Could have been created by e.g. `stream = bytearray(open('image.file', 'rb').read())`. Type `bytes` is supported in **Python 3 only**, because `bytes == str` in Python 2 and the method will interpret the stream as a filename.

Changed in version 1.14.13: `io.BytesIO` is now also supported.

__init__(*self*, *colorspace*, *width*, *height*, *samples*, *alpha*)

From plain pixels: Create a pixmap from `samples`. Each pixel must be represented by a number of bytes as controlled by the `colorspace` and `alpha` parameters. The origin of the resulting pixmap is `(0, 0)`. This method is useful when raw image data are provided by some other program – see *Collection of Recipes*.

Parameters

- `colorspace` (*Colorspace*) – Colorspace of image.
- `width` (*int*) – image width
- `height` (*int*) – image height
- `samples` (*bytes* , *bytearray* , *BytesIO*) – an area containing all pixels of the image. Must include alpha values if specified.

  Changed in version 1.14.13: (1) `io.BytesIO` can now also be used. (2) Data are now **copied** to the pixmap, so may safely be deleted or become unavailable.
- `alpha` (*bool*) – whether a transparency channel is included.

Note:

1. The following equation **must be true**: `(colorspace.n + alpha) * width * height == len(samples)`.
2. Starting with version 1.14.13, the samples data are **copied** to the pixmap.

__init__(*self*, *doc*, *xref*)

From a PDF image: Create a pixmap from an image **contained in PDF** `doc` identified by its *xref*. All pimap properties are set by the image. Have a look at extract-img1.py[77] and extract-img2.py[78] to see how this can be used to recover all of a PDF's images.

Parameters

- `doc` (*Document*) – an opened **PDF** document.
- `xref` (*int*) – the *xref* of an image object. For example, you can make a list of images used on a particular page with *Document.getPageImageList()*, which also shows the *xref* numbers of each image.

clearWith([*value*[, *irect*]])

Initialize the samples area.

Parameters

- `value` (*int*) – if specified, values from 0 to 255 are valid. Each color byte of each pixel will be set to this value, while alpha will be set to 255 (non-transparent) if present. If omitted, then all bytes (including any alpha) are cleared to `0x00`.
- `irect` (*irect_like*) – the area to be cleared. Omit to clear the whole pixmap. Can only be specified, if `value` is also specified.

tintWith(*red*, *green*, *blue*)

Colorize (tint) a pixmap with a color provided as an integer triple (red, green, blue). Only colorspaces *CS_GRAY* and *CS_RGB* are supported, others are ignored with a warning.

If the colorspace is *CS_GRAY*, `(red + green + blue)/3` will be taken as the tint value.

[77] https://github.com/pymupdf/PyMuPDF/tree/master/demo/extract-img1.py
[78] https://github.com/pymupdf/PyMuPDF/tree/master/demo/extract-img2.py

Parameters

- red (*int*) – red component.
- green (*int*) – green component.
- blue (*int*) – blue component.

gammaWith(*gamma*)

Apply a gamma factor to a pixmap, i.e. lighten or darken it. Pixmaps with colorspace None are ignored with a warning.

Parameters gamma (*float*) – gamma = 1.0 does nothing, gamma < 1.0 lightens, gamma > 1.0 darkens the image.

shrink(*n*)

Shrink the pixmap by dividing both, its width and height by 2^n.

Parameters n (*int*) – determines the new pixmap (samples) size. For example, a value of 2 divides width and height by 4 and thus results in a size of one 16^{th} of the original. Values less than 1 are ignored with a warning.

Note: Use this methods to reduce a pixmap's size retaining its proportion. The pixmap is changed "in place". If you want to keep original and also have more granular choices, use the resp. copy constructor above.

---

pixel(*x*, *y*)

New in version 1.14.5: Return the value of the pixel at location (x, y) (column, line).

Parameters

- x (*int*) – the column number of the pixel. Must be in range(pix.width).
- y (*int*) – the line number of the pixel, Must be in range(pix.height).

Return type list

Returns a list of color values and, potentially the alpha value. Its length and content depend on the pixmap's colorspace and the presence of an alpha. For RGBA pixmaps the result would e.g. be [r, g, b, a]. All items are integers in range(256).

setPixel(*x*, *y*, *color*)

New in version 1.14.7: Set the color of the pixel at location (x, y) (column, line).

Parameters

- x (*int*) – the column number of the pixel. Must be in range(pix.width).
- y (*int*) – the line number of the pixel. Must be in range(pix.height).
- color (*sequence*) – the desired color given as a sequence of integers in range(256). The length of the sequence must equal *Pixmap.n*, which includes any alpha byte.

setRect(*irect*, *color*)

New in version 1.14.8: Set the pixels of a rectangle to a color.

Parameters

- irect (*irect_like*) – the rectangle to be filled with the color. The actual area is the intersection of this parameter and *Pixmap.irect*. For an empty intersection (or an invalid parameter), no change will happen.

- color (*sequence*) – the desired color given as a sequence of integers in range(256). The length of the sequence must equal *Pixmap.n*, which includes any alpha byte.

Return type bool

Returns False if the rectangle was invalid or had an empty intersection with *Pixmap.irect*, else True.

Note:

1. This method is equivalent to *Pixmap.setPixel()* executed for each pixel in the rectangle, but is obviously **very much faster** if many pixels are involved.
2. This method can be used similar to *Pixmap.clearWith()* to initialize a pixmap with a certain color like this: pix.setRect(pix.irect, (255, 255, 0)) (RGB example, colors the complete pixmap with yellow).

---

setAlpha([*alphavalues*])

Change the alpha values. The pixmap must have an alpha channel.

Parameters alphavalues (*bytes*, *bytearray*, *BytesIO*) – the new alpha values. If provided, its length must be at least width * height. If omitted, all alpha values are set to 255 (no transparency).

Changed in version 1.14.13: io.BytesIO is now also supported.

invertIRect([*irect*])

Invert the color of all pixels in *IRect* irect. Will have no effect if colorspace is None.

Parameters irect (*irect_like*) – The area to be inverted. Omit to invert everything.

copyPixmap(*source*, *irect*)

Copy the irect part of the source pixmap into the corresponding area of this one. The two pixmaps may have different dimensions and can each have *CS_GRAY* or *CS_RGB* colorspaces, but they currently **must** have the same alpha property[84]. The copy mechanism automatically adjusts discrepancies between source and target like so:

If copying from *CS_GRAY* to *CS_RGB*, the source gray-shade value will be put into each of the three rgb component bytes. If the other way round, (r + g + b) / 3 will be taken as the gray-shade value of the target.

Between irect and the target pixmap's rectangle, an "intersection" is calculated at first. This takes into account the rectangle coordinates and the current attribute values source.x and source.y (which you are free to modify for this purpose). Then the corresponding data of this intersection are copied. If the intersection is empty, nothing will happen.

Parameters

- source (*Pixmap*) – source pixmap.
- irect (*irect_like*) – The area to be copied.

writeImage(*filename*, *output=None*)

Save pixmap as an image file. Depending on the output chosen, only some or all colorspaces are supported and different file extensions can be chosen. Please see the table below. Since MuPDF v1.10a the savealpha option is no longer supported and will be silently ignored.

Parameters

[84] To also set the alpha property, add an additional step to this method by dropping or adding an alpha channel to the result.

- `filename` (*str*) – The filename to save to. The filename's extension determines the image format, if not overriden by the output parameter.
- `output` (*str*) – The requested image format. The default is the filename's extension. If not recognized, `png` is assumed. For other possible values see *Supported Output Image Formats*.

`writePNG`(*filename*)
Equal to `pix.writeImage(filename, "png")`.

`getImageData`(*output="png"*)
New in version 1.14.5: Return the pixmap as a `bytes` memory object of the specified format – similar to *writeImage()*.

Parameters `output` (*str*) – The requested image format. The default is "png" for which this function equals *getPNGData()*. For other possible values see *Supported Output Image Formats*.

Return type bytes

`getPNGdata()`

`getPNGData()`
Equal to `pix.getImageData("png")`.

Return type bytes

`alpha`
Indicates whether the pixmap contains transparency information.

Type bool

`colorspace`
The colorspace of the pixmap. This value may be `None` if the image is to be treated as a so-called *image mask* or *stencil mask* (currently happens for extracted PDF document images only).

Type *Colorspace*

`stride`
Contains the length of one row of image data in *Pixmap.samples*. This is primarily used for calculation purposes. The following expressions are true:

- `len(samples) == height * stride`
- `width * n == stride`.

Type int

`irect`
Contains the *IRect* of the pixmap.

Type *IRect*

`samples`
The color and (if *Pixmap.alpha* is true) transparency values for all pixels. It is an area of `width * height * n` bytes. Each n bytes define one pixel. Each successive n bytes yield another pixel in scanline order. Subsequent scanlines follow each other with no padding. E.g. for an RGBA colorspace this means, `samples` is a sequence of bytes like `..., R, G, B, A, ...`, and the four byte values R, G, B, A define one pixel.

This area can be passed to other graphics libraries like PIL (Python Imaging Library) to do additional processing like saving the pixmap in other image formats.

Note:

- The underlying data is a typically **large** memory area from which a `bytes` copy is made for this attribute: for example an RGB-rendered letter page has a samples size of almost 1.4 MB. So consider assigning a new variable if you repeatedly use it.
- Any changes to the underlying data are available only after again accessing this attribute.

Type bytes

`size`
Contains `len(pixmap)`. This will generally equal `len(pix.samples)` plus some platform-specific value for defining other attributes of the object.

Type int

`width`

`w`
Width of the region in pixels.

Type int

`height`

`h`
Height of the region in pixels.

Type int

`x`
X-coordinate of top-left corner

Type int

`y`
Y-coordinate of top-left corner

Type int

`n`
Number of components per pixel. This number depends on colorspace and alpha. If colorspace is not `None` (stencil masks), then `Pixmap.n - Pixmap.aslpha == pixmap.colorspace.n` is true. If colorspace is `None`, then `n == alpha == 1`.

Type int

`xres`
Horizontal resolution in dpi (dots per inch).

Type int

`yres`
Vertical resolution in dpi.

Type int

`interpolate`
An information-only boolean flag set to `True` if the image will be drawn using "linear interpolation". If `False` "nearest neighbour sampling" will be used.

Type bool

5.12.1 Supported Input Image Formats

The following file types are supported as **input** to construct pixmaps: **BMP, JPEG, GIF, TIFF, JXR, JPX**, **PNG**, **PAM** and all of the **Portable Anymap** family (**PBM, PGM, PNM, PPM**). This support is two-fold:

1. Directly create a pixmap with `Pixmap(filename)` or `Pixmap(byterray)`. The pixmap will then have properties as determined by the image.
2. Open such files with `fitz.open(...)`. The result will then appear as a document containing one single page. Creating a pixmap of this page offers all the options available in this context: apply a matrix, choose colorspace and alpha, confine the pixmap to a clip area, etc.

SVG images are only supported via method 2 above, not directly as pixmaps. But remember: the result of this is a **raster image** as is always the case with pixmaps[79].

5.12.2 Supported Output Image Formats

A number of image **output** formats are supported. You have the option to either write an image directly to a file (*Pixmap.writeImage()*), or to generate a bytes object (*Pixmap.getImageData()*). Both methods accept a 3-letter string identifying the desired format (**Format** column below). Please note that not all combinations of pixmap colorspace, transparency support (alpha) and image format are possible.

Format	Colorspaces	alpha	Extensions	Description
pam	gray, rgb, cmyk	yes	.pam	Portable Arbitrary Map
pbm	gray, rgb	no	.pbm	Portable Bitmap
pgm	gray, rgb	no	.pgm	Portable Graymap
png	gray, rgb	yes	.png	Portable Network Graphics
pnm	gray, rgb	no	.pnm	Portable Anymap
ppm	gray, rgb	no	.ppm	Portable Pixmap
ps	gray, rgb, cmyk	no	.ps	Adobe PostScript Image
psd	gray, rgb, cmyk	yes	.psd	Adobe Photoshop Document

Note:

- Not all image file types are supported (or at least common) on all OS platforms. E.g. PAM and the Portable Anymap formats are rare or even unknown on Windows.
- Especially pertaining to CMYK colorspaces, you can always convert a CMYK pixmap to an RGB pixmap with `rgb_pix = fitz.Pixmap(fitz.csRGB, cmyk_pix)` and then save that in the desired format.
- As can be seen, MuPDF's image support range is different for input and output. Among those supported both ways, PNG is probably the most popular. We recommend using Pillow whenever you face a support gap.
- We also recommend using "ppm" formats as input to tkinter's `PhotoImage` method like this: `tkimg = tkinter.PhotoImage(data=pix.getImageData("ppm"))` (also see the tutorial). This is **very** fast (**60 times** faster than PNG) and will work under Python 2 or 3.

[79] If you need a **vector image** from the SVG, you must first convert it to a PDF. Try *Document.convertToPDF()*. If this is not not good enough, look for other SVG-to-PDF conversion tools like the Python packages svglib[80], CairoSVG[81], Uniconvertor[82] or the Java solution Apache Batik[83]. Have a look at our Wiki for more examples.

[80] https://pypi.org/project/svglib

[81] https://pypi.org/project/cairosvg

[82] https://sk1project.net/modules.php?name=Products&product=uniconvertor&op=download

[83] https://github.com/apache/batik

5.13 Point

`Point` represents a point in the plane, defined by its x and y coordinates.

Attribute / Method	Description
Point.distance_to()	calculate distance to point or rect
Point.norm()	the Euclidean norm
Point.transform()	transform point with a matrix
Point.abs_unit	same as unit, but positive coordinates
Point.unit	point coordinates divided by `abs(point)`
Point.x	the X-coordinate
Point.y	the Y-coordinate

Class API

`class Point`

`__init__`(*self*)

`__init__`(*self*, *x*, *y*)

`__init__`(*self*, *point*)

`__init__`(*self*, *sequence*)

Overloaded constructors.

Without parameters, `Point(0, 0)` will be created.

With another point specified, a **new copy** will be crated, "sequence" is a Python sequence of 2 numbers (see *Using Python Sequences as Arguments in PyMuPDF*).

Parameters

- `x` (*float*) – x coordinate of the point
- `y` (*float*) – y coordinate of the point

`distance_to`(*x*[, *unit*])

Calculate the distance to `x`, which may be *point_like* or *rect_like*. The distance is given in units of either pixels (default), inches, centimeters or millimeters.

Parameters

- `x` (*point_like*, *rect_like*) – to which to compute the distance.
- `unit` (*str*) – the unit to be measured in. One of "px", "in", "cm", "mm".

Return type float

Returns

the distance to `x`. If this is *rect_like*, then the distance

- is the length of the shortest line connecting to one of the rectangle sides
- is calculated to the **finite version** of it
- is zero if it **contains** the point

`norm()`
New in version 1.16.0: Return the Euclidean norm (the length) of the point as a vector. Equals result of function `abs()`.

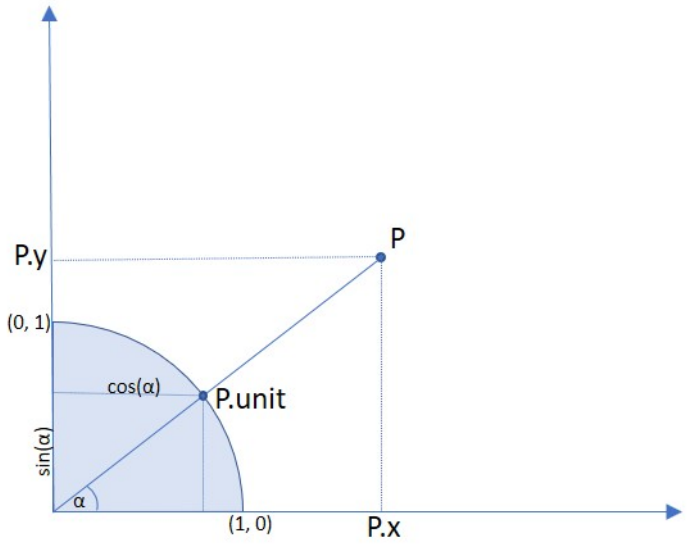
`transform(`*m*`)`

Apply a matrix to the point and replace it with the result.

Parameters `m` (*matrix_like*) – The matrix to be applied.

Return type *Point*

`unit`
Result of dividing each coordinate by `norm(point)`, the distance of the point to (0,0). This is a vector of length 1 pointing in the same direction as the point does. Its x, resp. y values are equal to the cosine, resp. sine of the angle this vector (and the point itself) has with the x axis.



Type *Point*

`abs_unit`
Same as *unit* above, replacing the coordinates with their absolute values.

Type *Point*

`x`
The x coordinate

Type float

`y`
The y coordinate

Type float

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
- Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.

5.14 Quad

Represents a four-sided mathematical shape (also called "quadrilateral" or "tetragon") in the plane, defined as a sequence of four *Point* objects ul, ur, ll, lr (conveniently called upper left, upper right, lower left, lower right).

Quads can **be obtained** as results of text search methods (*Page.searchFor()*), and they **are used** to define text marker annotations (see e.g. *Page.addSquigglyAnnot()* and friends), and in several draw methods (like `Page.drawQuad()` / *Shape.drawQuad()*, *Page.drawOval()* / :meth'Shape.drawQuad').

Note:

- If the corners of a rectangle are transformed with a **rotation**, **scale** or **translation** *Matrix*, then the resulting quad is **rectangular**, i.e. its corners again enclose angles of 90 degrees. Property *Quad.isRectangular* checks whether a quad can be thought of being the result of such an operation. This is not true for all matrices: e.g. shear matrices produce parallelograms, and non-invertible matrices deliver "degenerate" tetragons like triangles or lines.
- Attribute *Quad.rect* obtains the envelopping rectangle. Vice versa, rectangles now have attributes *Rect.quad*, resp. *IRect.quad* to obtain their respective tetragon versions.

Methods / Attributes	Short Description
Quad.transform()	transform with a matrix
Quad.ul	upper left point
Quad.ur	upper right point
Quad.ll	lower left point
Quad.lr	lower right point
Quad.isConvex	true if quad is a convex set
Quad.isEmpty	true if quad is an empty set
Quad.isRectangular	true if quad is a (rotated) rectangle
Quad.rect	smallest containing *Rect*
Quad.width	the longest width value
Quad.height	the longest height value

Class API

`class Quad`

`__init__`(*self*)

`__init__`(*self*, *ul*, *ur*, *ll*, *lr*)

`__init__`(*self*, *quad*)

`__init__`(*self*, *sequence*)

Overloaded constructors: "ul", "ur", "ll", "lr" stand for *point_like* objects (the four corners), "sequence" is a Python sequence with four *point_like* objects.

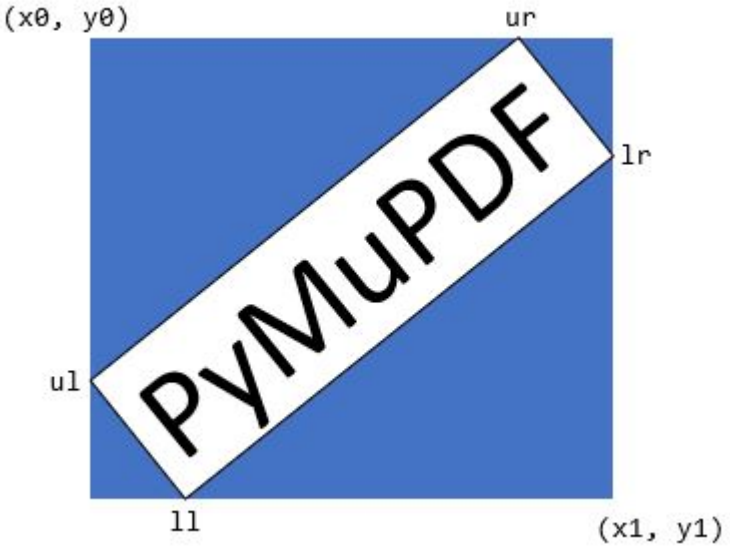
If "quad" is specified, the constructor creates a **new copy** of it.

Without parameters, a quad consisting of 4 copies of `Point(0, 0)` is created.

`transform`(*matrix*)
Modify the quadrilateral by transforming each of its corners with a matrix.

Parameters `matrix` (*matrix_like*) – the matrix.

`rect`
The smallest rectangle containing the quad, represented by the blue area in the following picture.



Type *Rect*

`ul`
Upper left point.

Type *Point*

`ur`
Upper right point.

Type *Point*

`ll`
Lower left point.

Type *Point*

`lr`
Lower right point.

Type *Point*

`isConvex`
New in version 1.16.1: True if all lines are contained in the quad which connect two points of the quad.

Type bool

`isEmpty`
True if enclosed area is zero, which means that all four points are on the same line. If this is false, the quad may still be degenerate or not look like a rectangle at all (triangles, parallelograms, trapezoids, ...).

Type bool

`isRectangular`
True if all angles are 90 degrees. This also implies that the area is **not empty** and **convex**.

Type bool

`width`
The maximum length of the top and the bottom side.

Type float

`height`
The maximum length of the left and the right side.

Type float

5.14.1 Remark

This class adheres to the sequence protocol, so components can be dealt with via their indices, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.

We are still in process to extend algebraic operations to quads. Multiplication and division with / by numbers and matrices are already defined. Addition, subtraction and any unary operations may follow when we see an actual need.

5.15 Rect

`Rect` represents a rectangle defined by four floating point numbers x0, y0, x1, y1. They are treated as being coordinates of two diagonally opposite points. The first two numbers are regarded as the "top left" corner $P_{x0,y0}$ and $P_{x1,y1}$ as the "bottom right" one. However, these two properties need not coincide with their intuitive meanings – read on.

The following remarks are also valid for *IRect* objects:

- Rectangle borders are always parallel to the respective X- and Y-axes.
- The constructing points can be anywhere in the plane – they need not even be different, and e.g. "top left" need not be the geometrical "north-western" point.
- For any given quadruple of numbers, the geometrically "same" rectangle can be defined in (up to) four different ways: Rect($P_{x0,y0}$, $P_{x1,y1}$), Rect($P_{x1,y1}$, $P_{x0,y0}$), Rect($P_{x0,y1}$, $P_{x1,y0}$), and Rect($P_{x1,y0}$, $P_{x0,y1}$).

Hence some useful classification:

- A rectangle is called **finite** if `x0 <= x1` and `y0 <= y1` (i.e. the bottom right point is "south-eastern" to the top left one), otherwise **infinite**. Of the four alternatives above, **only one** is finite (disregarding degenerate cases). Please take into account, that in MuPDF's coordinate system the y-axis is oriented from **top to bottom**.
- A rectangle is called **empty** if `x0 = x1` or `y0 = y1`, i.e. if its area is zero.

Note: It sounds like a paradox: a rectangle can be both, infinite **and** empty ...

Methods / Attributes	Short Description
Rect.contains()	checks containment of another object
Rect.getArea()	calculate rectangle area
Rect.getRectArea()	calculate rectangle area
Rect.includePoint()	enlarge rectangle to also contain a point
Rect.includeRect()	enlarge rectangle to also contain another one
Rect.intersect()	common part with another rectangle
Rect.intersects()	checks for non-empty intersections
Rect.norm()	the Euclidean norm
Rect.normalize()	makes a rectangle finite
Rect.round()	create smallest *IRect* containing rectangle
Rect.transform()	transform rectangle with a matrix
Rect.bottom_left	bottom left point, synonym `bl`
Rect.bottom_right	bottom right point, synonym `br`
Rect.height	rectangle height
Rect.irect	equals result of method `round()`
Rect.isEmpty	whether rectangle is empty
Rect.isInfinite	whether rectangle is infinite
Rect.top_left	top left point, synonym `tl`
Rect.top_right	top_right point, synonym `tr`
Rect.quad	*Quad* made from rectangle corners
Rect.width	rectangle width
Rect.x0	top left corner's X-coordinate
Rect.x1	bottom right corner's X-coordinate
Rect.y0	top left corner's Y-coordinate
Rect.y1	bottom right corner's Y-coordinate

Class API

`class Rect`

`__init__`(*self*)

`__init__`(*self*, *x0*, *y0*, *x1*, *y1*)

`__init__`(*self*, *top_left*, *bottom_right*)

`__init__`(*self*, *top_left*, *x1*, *y1*)

`__init__`(*self*, *x0*, *y0*, *bottom_right*)

`__init__`(*self*, *rect*)

`__init__`(*self*, *sequence*)

Overloaded constructors: `top_left`, `bottom_right` stand for *point_like* objects, "sequence" is a Python sequence type of 4 numbers (see *Using Python Sequences as Arguments in PyMuPDF*), "rect" means another *rect_like*, while the other parameters mean coordinates.

If "rect" is specified, the constructor creates a **new copy** of it.

Without parameters, the empty rectangle `Rect(0.0, 0.0, 0.0, 0.0)` is created.

`round()`

Creates the smallest containing *IRect*, This is **not** the same as simply rounding the rectangle's edges: The top left corner is rounded upwards and left while the bottom right corner is rounded downwards and to the right.

```
>>> fitz.Rect(0.5, -0.01, 123.88, 455.123456).round()
IRect(0, -1, 124, 456)
```

1. If the rectangle is **infinite**, the "normalized" (finite) version of it will be taken. The result of this method is always a finite `IRect`.
2. If the rectangle is **empty**, the result is also empty.
3. **Possible paradox:** The result may be empty, **even if** the rectangle is **not** empty! In such cases, the result obviously does **not** contain the rectangle. This is because MuPDF's algorithm allows for a small tolerance (1e-3). Example:

```
>>> r = fitz.Rect(100, 100, 200, 100.001)
>>> r.isEmpty  # rect is NOT empty
False
>>> r.round()  # but its irect IS empty!
fitz.IRect(100, 100, 200, 100)
>>> r.round().isEmpty
True
```

Return type *IRect*

`transform(`*m*`)`

Transforms the rectangle with a matrix and **replaces the original**. If the rectangle is empty or infinite, this is a no-operation.

Parameters `m` (*Matrix*) – The matrix for the transformation.

Return type `Rect`

Returns the smallest rectangle that contains the transformed original.

`intersect(`*r*`)`

The intersection (common rectangular area) of the current rectangle and `r` is calculated and **replaces the current** rectangle. If either rectangle is empty, the result is also empty. If `r` is infinite, this is a no-operation.

Parameters `r` (*Rect*) – Second rectangle

`includeRect(`*r*`)`

The smallest rectangle containing the current one and `r` is calculated and **replaces the current** one. If either rectangle is infinite, the result is also infinite. If one is empty, the other one will be taken as the result.

Parameters `r` (*Rect*) – Second rectangle

`includePoint(`*p*`)`

The smallest rectangle containing the current one and point `p` is calculated and **replaces the current** one. **Infinite rectangles remain unchanged.** To create a rectangle containing a series of points, start with (the empty) `fitz.Rect(p1, p1)` and successively perform `includePoint` operations for the other points.

Parameters `p` (*Point*) – Point to include.

`getRectArea(`[*unit*]`)`

`getArea(`[*unit*]`)`

Calculate the area of the rectangle and, with no parameter, equals `abs(rect)`. Like an empty

rectangle, the area of an infinite rectangle is also zero. So, at least one of `fitz.Rect(p1, p2)` and `fitz.Rect(p2, p1)` has a zero area.

Parameters `unit` (*str*) – Specify required unit: respective squares of `px` (pixels, default), `in` (inches), `cm` (centimeters), or `mm` (millimeters).

Return type float

`contains(x)`
Checks whether `x` is contained in the rectangle. It may be an `IRect`, `Rect`, `Point` or number. If `x` is an empty rectangle, this is always true. If the rectangle is empty this is always `False` for all non-empty rectangles and for all points. If `x` is a number, it will be checked against the four components. `x in rect` and `rect.contains(x)` are equivalent.

Parameters `x` (*IRect* or *Rect* or *Point* or number) – the object to check.

Return type bool

`intersects(r)`
Checks whether the rectangle and a *rect_like* "r" contain a common non-empty *Rect*. This will always be `False` if either is infinite or empty.

Parameters `r` (*rect_like*) – the rectangle to check.

Return type bool

`norm()`
New in version 1.16.0: Return the Euclidean norm of the rectangle treated as a vector of four numbers.

`normalize()`
Replace the rectangle with its finite version. This is done by shuffling the rectangle corners. After completion of this method, the bottom right corner will indeed be south-eastern to the top left one.

`irect`
Equals result of method `round()`.

`top_left`

`tl`
Equals `Point(x0, y0)`.

Type *Point*

`top_right`

`tr`
Equals `Point(x1, y0)`.

Type *Point*

`bottom_left`

`bl`
Equals `Point(x0, y1)`.

Type *Point*

`bottom_right`

`br`
Equals `Point(x1, y1)`.

Type *Point*

`quad`
The quadrilateral `Quad(rect.tl, rect.tr, rect.bl, rect.br)`.

Type *Quad*

`width`
Width of the rectangle. Equals `abs(x1 - x0)`.

Return type float

`height`
Height of the rectangle. Equals `abs(y1 - y0)`.

Return type float

`x0`
X-coordinate of the left corners.

Type float

`y0`
Y-coordinate of the top corners.

Type float

`x1`
X-coordinate of the right corners.

Type float

`y1`
Y-coordinate of the bottom corners.

Type float

`isInfinite`
`True` if rectangle is infinite, `False` otherwise.

Type bool

`isEmpty`
`True` if rectangle is empty, `False` otherwise.

Type bool

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
- Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.

5.16 Shape

This class allows creating interconnected graphical elements on a PDF page. Its methods have the same meaning and name as the corresponding *Page* methods.

In fact, each *Page* draw method is just a convenience wrapper for (1) one shape draw method, (2) the `finish()` method, and (3) the `commit()` method. For page text insertion, only the `commit()` method is

invoked. If many draw and text operations are executed for a page, you should always consider using a Shape object.

Several draw methods can be executed in a row and each one of them will contribute to one drawing. Once the drawing is complete, the `finish()` method must be invoked to apply color, dashing, width, morphing and other attributes.

Draw methods of this class (and `insertTextbox()`) are logging the area they are covering in a rectangle (*Shape.rect*). This property can for instance be used to set *Page.CropBox*.

Text insertions `insertText()` and `insertTextbox()` implicitely execute a "finish" and therefore only require `commit()` to become effective. As a consequence, both include parameters for controlling prperties like colors, etc.

Method / Attribute	Description
Shape.commit()	update the page's contents
Shape.drawBezier()	draw a cubic Bezier curve
Shape.drawCircle()	draw a circle around a point
Shape.drawCurve()	draw a cubic Bezier using one helper point
Shape.drawLine()	draw a line
Shape.drawOval()	draw an ellipse
Shape.drawPolyline()	connect a sequence of points
Shape.drawQuad()	draw a quadrilateral
Shape.drawRect()	draw a rectangle
Shape.drawSector()	draw a circular sector or piece of pie
Shape.drawSquiggle()	draw a squiggly line
Shape.drawZigzag()	draw a zigzag line
Shape.finish()	finish a set of draw commands
Shape.insertText()	insert text lines
Shape.insertTextbox()	fit text into a rectangle
Shape.doc	stores the page's document
Shape.draw_cont	draw commands since last `finish()`
Shape.height	stores the page's height
Shape.lastPoint	stores the current point
Shape.page	stores the owning page
Shape.rect	rectangle surrounding drawings
Shape.text_cont	accumulated text insertions
Shape.totalcont	accumulated string to be stored in *contents*
Shape.width	stores the page's width

Class API

`class Shape`

`__init__`(*self*, *page*)

Create a new drawing. During importing PyMuPDF, the `fitz.Page` object is being given the convenience method `newShape()` to construct a `Shape` object. During instantiation, a check will be made whether we do have a PDF page. An exception is otherwise raised.

Parameters `page` (*Page*) – an existing page of a PDF document.

`drawLine`(*p1*, *p2*)

Draw a line from *point_like* objects `p1` to `p2`.

Parameters

- p1 (*point_like*) – starting point
- p2 (*point_like*) – end point

Return type *Point*

Returns the end point, p2.

drawSquiggle(*p1*, *p2*, *breadth=2*)

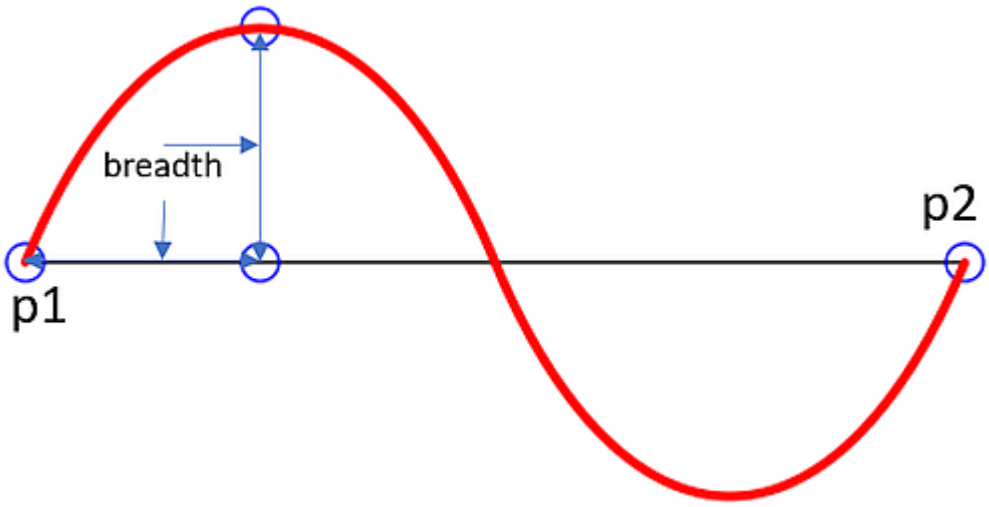
Draw a squiggly (wavy, undulated) line from *point_like* objects p1 to p2. An integer number of full wave periods will always be drawn, one period having a length of `4 * breadth`. The breadth parameter will be adjusted as necessary to meet this condition. The drawn line will always turn "left" when leaving p1 and always join p2 from the "right".

Parameters

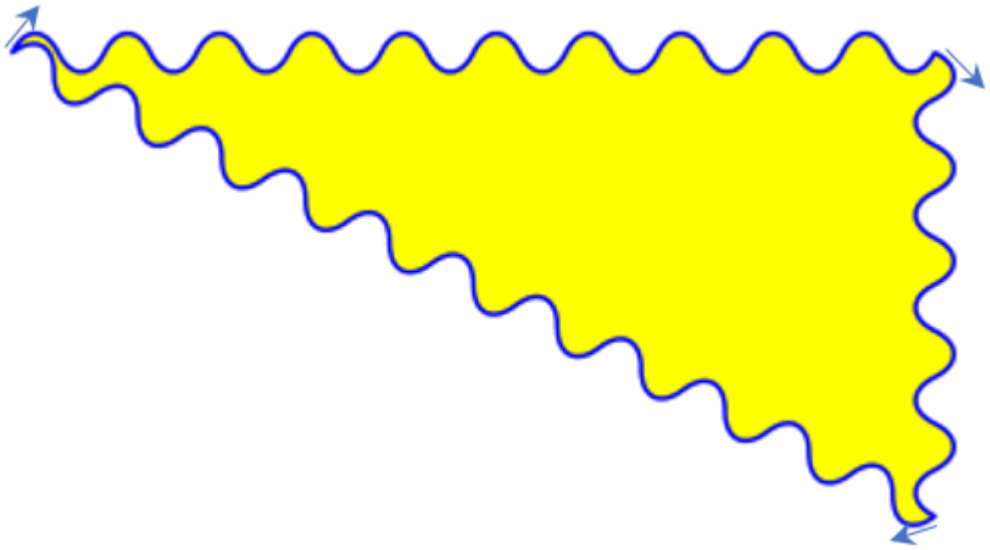
- p1 (*point_like*) – starting point
- p2 (*point_like*) – end point
- `breadth` (*float*) – the amplitude of each wave. The condition `2 * breadth < abs(p2 - p1)` must be true to fit in at least one wave. See the following picture, which shows two points connected by one full period.

Return type *Point*

Returns the end point, p2.



Here is an example of three connected lines, forming a closed, filled triangle. Little arrows indicate the stroking direction.



Note: Waves drawn are **not** trigonometric (sine / cosine). If you need that, have a look at draw-sines.py[85].

`drawZigzag`(*p1*, *p2*, *breadth=2*)

Draw a zigzag line from *point_like* objects `p1` to `p2`. An integer number of full zigzag periods will always be drawn, one period having a length of `4 * breadth`. The breadth parameter will be adjusted to meet this condition. The drawn line will always turn "left" when leaving `p1` and always join `p2` from the "right".

Parameters

- `p1` (*point_like*) – starting point
- `p2` (*point_like*) – end point
- `breadth` (*float*) – the amplitude of the movement. The condition `2 * breadth < abs(p2 - p1)` must be true to fit in at least one period.

Return type *Point*

Returns the end point, `p2`.

`drawPolyline`(*points*)

Draw several connected lines between points contained in the sequence `points`. This can be used for creating arbitrary polygons by setting the last item equal to the first one.

Parameters `points` (*sequence*) – a sequence of *point_like* objects. Its length must at least be 2 (in which case it is equivalent to `drawLine()`).

Return type *Point*

Returns `points[-1]` – the last point in the argument sequence.

`drawBezier`(*p1*, *p2*, *p3*, *p4*)

Draw a standard cubic Bezier curve from `p1` to `p4`, using `p2` and `p3` as control points.

All arguments are *point_like* s.

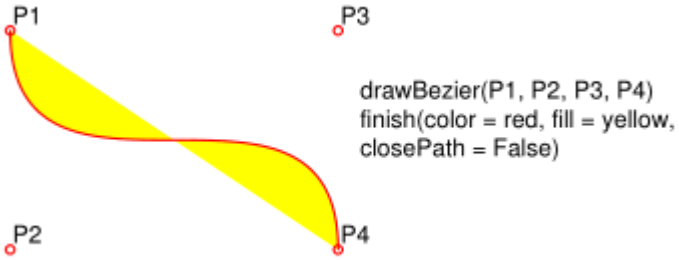
[85] https://github.com/pymupdf/PyMuPDF/blob/master/demo/draw-sines.py

Return type *Point*

Returns the end point, `p4`.

Note: The points do not need to be different – experiment a bit with some of them being equal!

Example:



`drawOval`(*tetra*)

Draw an "ellipse" inside the given tetragon (quadrilateral). If it is a square, a regular circle is drawn, a general rectangle will result in an ellipse. If a quadrilateral is used instead, a plethora of shapes can be the result.

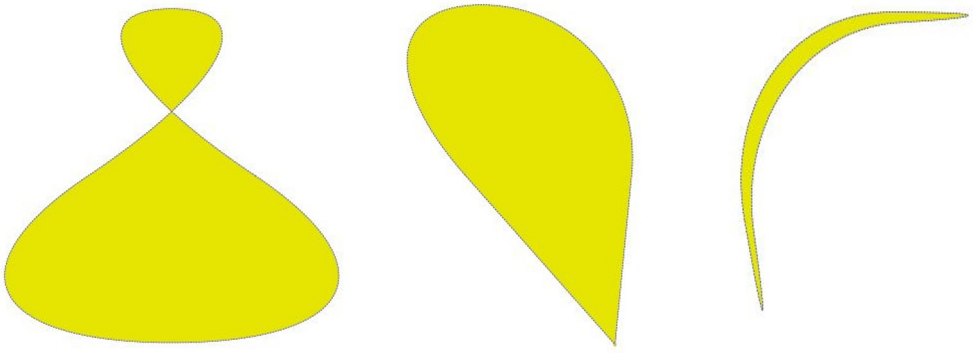
The drawing starts and ends at the middle point of the line connecting bottom-left and top-left corners in an anti-clockwise movement.

Parameters `tetra` (*rect_like*, *quad_like*) – *rect_like* or *quad_like*.

Changed in version 1.14.5: tetragons are now also supported.

Return type *Point*

Returns the middle point of line from `rect.bl` to `rect.tl`, or from `quad.ll` to `quad.ul`, respectively. Look at just a few examples here, or at the `quad-show?.py` scripts in the PyMuPDF-Utilities repository.



`drawCircle`(*center*, *radius*)

Draw a circle given its center and radius. The drawing starts and ends at point `center - (radius, 0)` in an anti-clockwise movement. This corresponds to the middle point of the enclosing rectangle's left side.

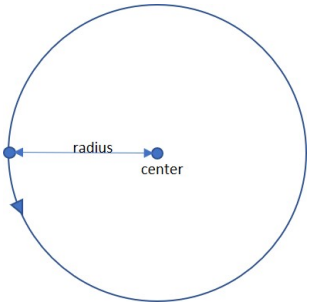
The method is a shortcut for `drawSector(center, start, 360, fullSector=False)`. To draw a circle in a clockwise movement, change the sign of the degree.

Parameters

- `center` (*point_like*) – the center of the circle.
- `radius` (*float*) – the radius of the circle. Must be positive.

Return type *Point*

Returns `center - (radius, 0)`.



`drawCurve`(*p1*, *p2*, *p3*)

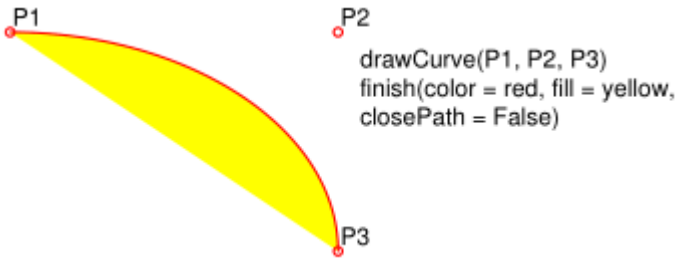
A special case of `drawBezier()`: Draw a cubic Bezier curve from `p1` to `p3`. On each of the two lines from `p1` to `p2` and from `p2` to `p3` one control point is generated. This guaranties that the curve's curvature does not change its sign. If these two connecting lines intersect with an angle of 90 degrees, then the resulting curve is a quarter ellipse (or quarter circle, if of same length) circumference.

All arguments are *point_like*.

Return type *Point*

Returns the end point, `p3`.

Example: a filled quarter ellipse segment.



`drawSector`(*center*, *point*, *angle*, *fullSector=True*)

Draw a circular sector, optionally connecting the arc to the circle's center (like a piece of pie).

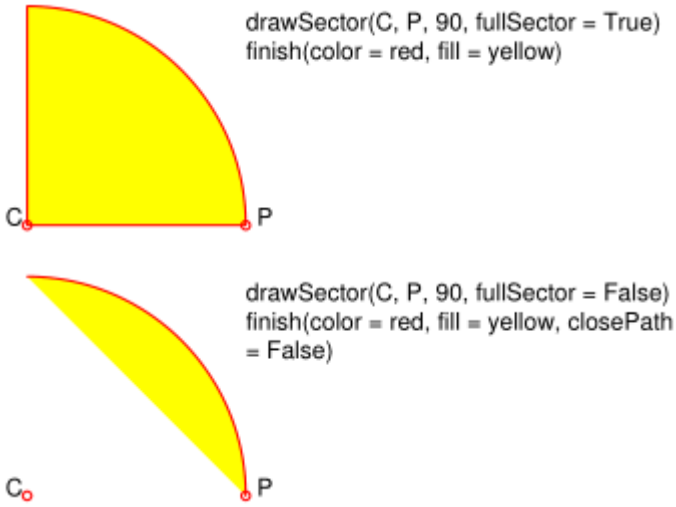
Parameters

- `center` (*point_like*) – the center of the circle.
- `point` (*point_like*) – one of the two end points of the pie's arc segment. The other one is calculated from the `angle`.
- `angle` (*float*) – the angle of the sector in degrees. Used to calculate the other end point of the arc. Depending on its sign, the arc is drawn anti-clockwise (postive) or clockwise.
- `fullSector` (*bool*) – whether to draw connecting lines from the ends of the arc to the circle center. If a fill color is specified, the full "pie" is colored, otherwise just the sector.

Returns the other end point of the arc. Can be used as starting point for a following invocation to create logically connected pies charts.

Return type *Point*

Examples:



`drawRect`(*rect*)

Draw a rectangle. The drawing starts and ends at the top-left corner in an anti-clockwise movement.

Parameters `rect` (*rect_like*) – where to put the rectangle on the page.

Return type *Point*

Returns top-left corner of the rectangle.

`drawQuad`(*quad*)

Draw a quadrilateral. The drawing starts and ends at the top-left corner (*Quad.ul*) in an anti-clockwise movement. It invokes *drawPolyline()* with the argument `[ul, ll, lr, ur, ul]`.

Parameters `quad` (*quad_like*) – where to put the tetragon on the page.

Return type *Point*

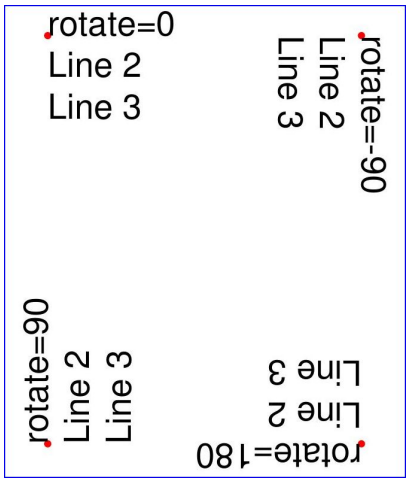
Returns *Quad.ul*.

`insertText`(*point*, *text*, *fontsize=11*, *fontname="helv"*, *fontfile=None*, *set_simple=False*, *encoding=TEXT_ENCODING_LATIN*, *color=None*, *fill=None*, *render_mode=0*, *border_width=1*, *rotate=0*, *morph=None*)

Insert text lines start at `point`.

Parameters

- `point` (*point_like*) – the bottom-left position of the first character of `text` in pixels. It is important to understand, how this works in conjunction with the `rotate` parameter. Please have a look at the following picture. The small red dots indicate the positions of `point` in each of the four possible cases.



- `text` (*str/sequence*) – the text to be inserted. May be specified as either a string type or as a sequence type. For sequences, or strings containing line breaks `\n`, several lines will be inserted. No care will be taken if lines are too wide, but the number of inserted lines will be limited by "vertical" space on the page (in the sense of reading direction as established by the `rotate` parameter). Any rest of `text` is discarded – the return code however contains the number of inserted lines.
- `rotate` (*int*) – determines whether to rotate the text. Acceptable values are multiples of 90 degrees. Default is 0 (no rotation), meaning horizontal text lines oriented from left to right. 180 means text is shown upside down from **right to left**. 90 means anti-clockwise rotation, text running **upwards**. 270 (or -90) means clockwise rotation, text running **downwards**. In any case, `point` specifies the bottom-left coordinates of the first character's rectangle. Multiple lines, if present, always follow the reading direction established by this parameter. So line 2 is located **above** line 1 in case of `rotate = 180`, etc.

Return type int

Returns number of lines inserted.

For a description of the other parameters see *Common Parameters*.

`insertTextbox`(*rect*, *buffer*, *fontsize=11*, *fontname="helv"*, *fontfile=None*, *set_simple=False*, *encoding=TEXT_ENCODING_LATIN*, *color=None*, *fill=None*, *render_mode=0*, *border_width=1*, *expandtabs=8*, *align=TEXT_ALIGN_LEFT*, *rotate=0*, *morph=None*)

PDF only: Insert text into the specified rectangle. The text will be split into lines and words and then filled into the available space, starting from one of the four rectangle corners, which depends on `rotate`. Line feeds will be respected as well as multiple spaces will be.

Parameters

- `rect` (*rect_like*) – the area to use. It must be finite and not empty.
- `buffer` (*str/sequence*) – the text to be inserted. Must be specified as a string or a sequence of strings. Line breaks are respected also when occurring in a sequence entry.
- `align` (*int*) – align each text line. Default is 0 (left). Centered, right and justified are the other supported options, see *Text Alignment*. Please note that the effect of parameter value `TEXT_ALIGN_JUSTIFY` is only achievable with "simple" (single-byte) fonts (including the *PDF Base 14 Fonts*). Refer to *Adobe PDF Reference 1.7*,

section 5.2.2, page 399.

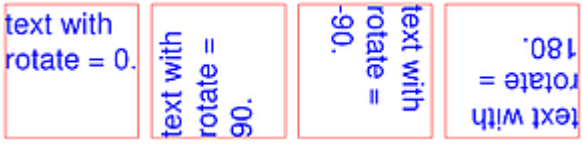
- `expandtabs` (*int*) – controls handling of tab characters `\t` using the `string.expandtabs()` method **per each line**.
- `rotate` (*int*) – requests text to be rotated in the rectangle. This value must be a multiple of 90 degrees. Default is 0 (no rotation). Effectively, four different values are processed: 0, 90, 180 and 270 (= -90), each causing the text to start in a different rectangle corner. Bottom-left is 90, bottom-right is 180, and -90 / 270 is top-right. See the example how text is filled in a rectangle. This argument takes precedence over morphing. See the second example, which shows text first rotated left by 90 degrees and then the whole rectangle rotated clockwise around is lower left corner.

Return type float

Returns

If positive or zero: successful execution. The value returned is the unused rectangle line space in pixels. This may safely be ignored – or be used to optimize the rectangle, position subsequent items, etc.

If negative: no execution. The value returned is the space deficit to store text lines. Enlarge rectangle, decrease `fontsize`, decrease text amount, etc.



For a description of the other parameters see *Common Parameters*.

`finish`(*width=1*, *color=None*, *fill=None*, *lineCap=0*, *lineJoin=0*, *dashes=None*, *closePath=True*, *even_odd=False*, *morph=(pivot*, *matrix)*)

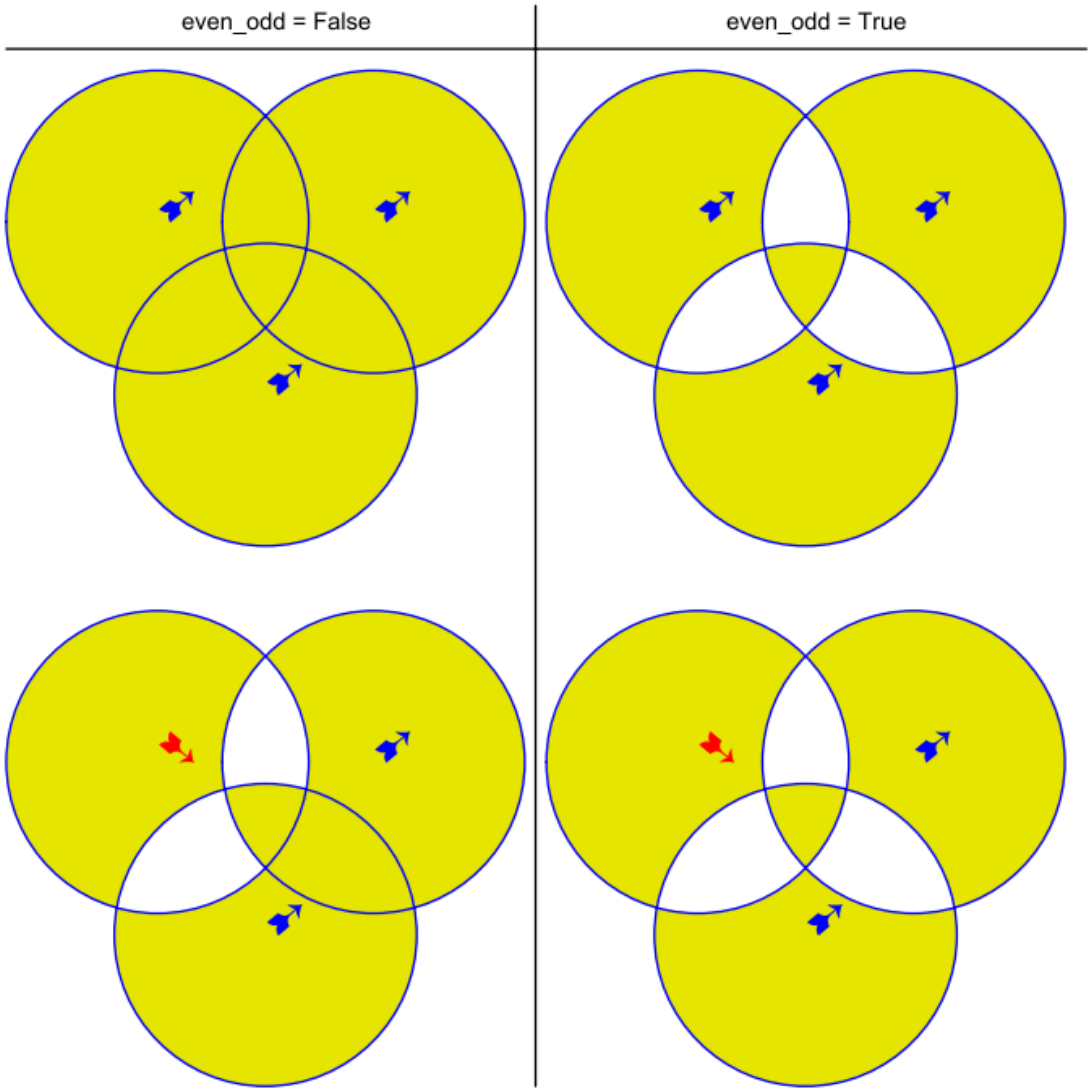
Finish a set of `draw*()` methods by applying *Common Parameters* to all of them. This method also supports morphing the resulting compound drawing using a pivotal *Point*.

Parameters

- `morph` (*sequence*) – morph the text or the compound drawing around some arbitrary pivotal *Point* `pivot` by applying *Matrix* `matrix` to it. This implies that `pivot` is a **fixed point** of this operation. Default is no morphing (`None`). The matrix can contain any values in its first 4 components, `matrix.e == matrix.f == 0` must

be true, however. This means that any combination of scaling, shearing, rotating, flipping, etc. is possible, but translations are not.

- `even_odd` (*bool*) – request the **"even-odd rule"** for filling operations. Default is `False`, so that the **"nonzero winding number rule"** is used. These rules are alternative methods to apply the fill color where areas overlap. Only with fairly complex shapes a different behavior is to be expected with these rules. For an in-depth explanation, see *Adobe PDF Reference 1.7*, pp. 232 ff. Here is an example to demonstrate the difference.



Note: For each pixel in a drawing the following will happen:

1. Rule **"even-odd"** counts, how many areas are overlapping at a pixel. If this count is **odd** the pixel is regarded **inside**, if it is **even**, the pixel is **outside**.
2. Default rule **"nonzero winding"** also looks at the orientation of overlapping areas: it **adds 1** if an area is drawn anit-clockwise and it **subtracts 1** for clockwise areas. If the result is zero, the pixel is regarded **outside**, pixels with a non-zero count are **inside**.

In the top two shapes, three circles are drawn in standard manner (anti-clockwise, look at the

arrows). The lower two shapes contain one (top-left) circle drawn clockwise. As can be seen, area orientation is irrelevant for the even-odd rule.

`commit(`*overlay=True*`)`

Update the page's *contents* with the accumulated draw commands and text insertions. If a `Shape` is not committed, the page will not be changed.

The method will reset attributes *Shape.rect*, *lastPoint*, *draw_cont*, *text_cont* and *totalcont*. Afterwards, the shape object can be reused for the **same page**.

Parameters `overlay` (*bool*) – determine whether to put content in foreground (default) or background. Relevant only, if the page already has a non-empty *contents* object.

`doc`

For reference only: the page's document.

Type *Document*

`page`

For reference only: the owning page.

Type *Page*

`height`

Copy of the page's height

Type float

`width`

Copy of the page's width.

Type float

`draw_cont`

Accumulated command buffer for **draw methods** since last finish.

Type str

`text_cont`

Accumulated text buffer. All **text insertions** go here. On *commit()* this buffer will be appended to *totalcont*, so that text will never be covered by drawings in the same Shape.

Type str

`rect`

Rectangle surrounding drawings. This attribute is at your disposal and may be changed at any time. Its value is set to `None` when a shape is created or committed. Every `draw*` method, and *Shape.insertTextbox()* update this property (i.e. **enlarge** the rectangle as needed). **Morphing** operations, however (*Shape.finish()*, *Shape.insertTextbox()*) are ignored.

A typical use of this attribute would be setting *Page.CropBox* to this value, when you are creating shapes for later or external use. If you have not manipulated the attribute yourself, it should reflect a rectangle that contains all drawings so far.

If you have used morphing and need a rectangle containing the morphed objects, use the following code:

```
>>> # assuming ...
>>> morph = (point, matrix)
>>> # ... recalculate the shape rectangle like so:
>>> shape.rect = (shape.rect - fitz.Rect(point, point)) * ~matrix + fitz.Rect(point,␣
↪point)
```

(continues on next page)

(continued from previous page)

Type *Rect*

`totalcont`
Total accumulated command buffer for draws and text insertions. This will be used by *Shape.commit()*.

Type str

`lastPoint`
For reference only: the current point of the drawing path. It is `None` at `Shape` creation and after each `finish()` and `commit()`.

Type *Point*

5.16.1 Usage

A drawing object is constructed by `shape = page.newShape()`. After this, as many draw, finish and text insertions methods as required may follow. Each sequence of draws must be finished before the drawing is committed. The overall coding pattern looks like this:

```
>>> shape = page.newShape()
>>> shape.draw1(...)
>>> shape.draw2(...)
>>> ...
>>> shape.finish(width=..., color=..., fill=..., morph=...)
>>> shape.draw3(...)
>>> shape.draw4(...)
>>> ...
>>> shape.finish(width=..., color=..., fill=..., morph=...)
>>> ...
>>> shape.insertText*
>>> ...
>>> shape.commit()
>>> ....
```

Note:

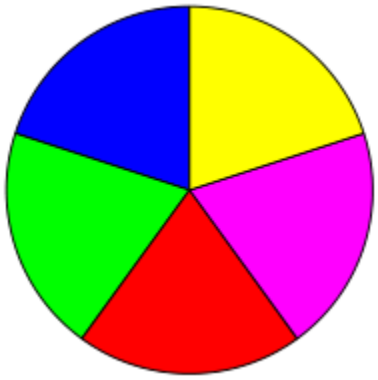
1. Each `finish()` combines the preceding draws into one logical shape, giving it common colors, line width, morphing, etc. If `closePath` is specified, it will also connect the end point of the last draw with the starting point of the first one.
2. To successfully create compound graphics, let each draw method use the end point of the previous one as its starting point. In the above pseudo code, `draw2` should hence use the returned *Point* of `draw1` as its starting point. Failing to do so, would automatically start a new path and `finish()` may not work as expected (but it won't complain either).
3. Text insertions may occur anywhere before the commit (they neither touch *Shape.draw_cont* nor *Shape.lastPoint*). They are appended to `Shape.totalcont` directly, whereas draws will be appended by `Shape.finish`.
4. Each `commit` takes all text insertions and shapes and places them in foreground or background on the page – thus providing a way to control graphical layers.
5. **Only** `commit` **will update** the page's contents, the other methods are basically string manipulations.

5.16.2 Examples

1. Create a full circle of pieces of pie in different colors:

```
>>> shape  = page.newShape()       # start a new shape
>>> cols = (...)                   # a sequence of RGB color triples
>>> pieces = len(cols)             # number of pieces to draw
>>> beta = 360. / pieces           # angle of each piece of pie
>>> center = fitz.Point(...)       # center of the pie
>>> p0     = fitz.Point(...)       # starting point
>>> for i in range(pieces):
        p0 = shape.drawSector(center, p0, beta,
                              fullSector=True) # draw piece
        # now fill it but do not connect ends of the arc
        shape.finish(fill=cols[i], closePath=False)
>>> shape.commit()                 # update the page
```

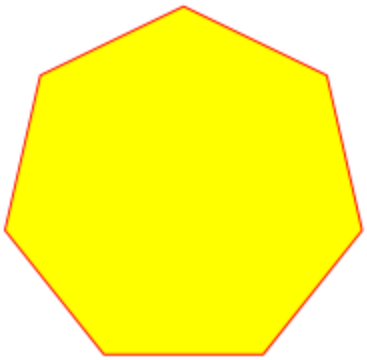
Here is an example for 5 colors:



2. Create a regular n-edged polygon (fill yellow, red border). We use `drawSector()` only to calculate the points on the circumference, and empty the draw command buffer before drawing the polygon:

```
>>> shape  = page.newShape()       # start a new shape
>>> beta = -360.0 / n              # our angle, drawn clockwise
>>> center = fitz.Point(...)       # center of circle
>>> p0     = fitz.Point(...)       # start here (1st edge)
>>> points = [p0]                  # store polygon edges
>>> for i in range(n):             # calculate the edges
        p0 = shape.drawSector(center, p0, beta)
        points.append(p0)
>>> shape.draw_cont = ""           # do not draw the circle sectors
>>> shape.drawPolyline(points)     # draw the polygon
>>> shape.finish(color=(1,0,0), fill=(1,1,0), closePath=False)
>>> shape.commit()
```

Here is the polygon for n = 7:



5.16.3 Common Parameters

fontname (*str*)

In general, there are three options:

1. Use one of the standard *PDF Base 14 Fonts*. In this case, `fontfile` **must not** be specified and `"Helvetica"` is used if this parameter is omitted, too.
2. Choose a font already in use by the page. Then specify its **reference** name prefixed with a slash "/", see example below.
3. Specify a font file present on your system. In this case choose an arbitrary, but new name for this parameter (without "/" prefix).

If inserted text should re-use one of the page's fonts, use its reference name appearing in `getFontList()` like so:

Suppose the font list has the entry `[1024, 0, 'Type1', 'CJXQIC+NimbusMonL-Bold', 'R366']`, then specify `fontname = "/R366", fontfile = None` to use font `CJXQIC+NimbusMonL-Bold`.

---

fontfile (*str*)

File path of a font existing on your computer. If you specify `fontfile`, make sure you use a `fontname` **not occurring** in the above list. This new font will be embedded in the PDF upon `doc.save()`. Similar to new images, a font file will be embedded only once. A table of MD5 codes for the binary font contents is used to ensure this.

---

set_simple (*bool*)

Fonts installed from files are installed as **Type0** fonts by default. If you want to use 1-byte characters only, set this to true. This setting cannot be reverted. Subsequent changes are ignored.

---

fontsize (*float*)

Font size of text. This also determines the line height as `fontsize * 1.2`.

---

dashes (*str*)

Causes lines to be dashed. A continuous line with no dashes is drawn with `"[]0"` or `None`. For (the rather complex) details on how to achieve dashing effects, see *Adobe PDF Reference 1.7*, page 217. Simple versions look like `"[3 4]"`, which means dashes of 3 and gaps of 4 pixels length follow each other. `"[3 3]"` and `"[3]"` do the same thing.

---

color / fill (*list, tuple*)

Line and fill colors can be specified as tuples or list of of floats from 0 to 1. These sequences must have a length of 1 (GRAY), 3 (RGB) or 4 (CMYK). For GRAY colorspace, a single float instead of the unwieldy `(float,)` tuple spec is also accepted.

To simplify color specification, method `getColor()` in `fitz.utils` may be used to get predefined RGB color triples by name. It accepts a string as the name of the color and returns the corresponding triple. The method knows over 540 color names – see section *Color Database*.

---

border_width (*float*)

Set the border width for text insertions. New in v1.14.9. Relevant only if the render mode argument is used with a value greater zero.

---

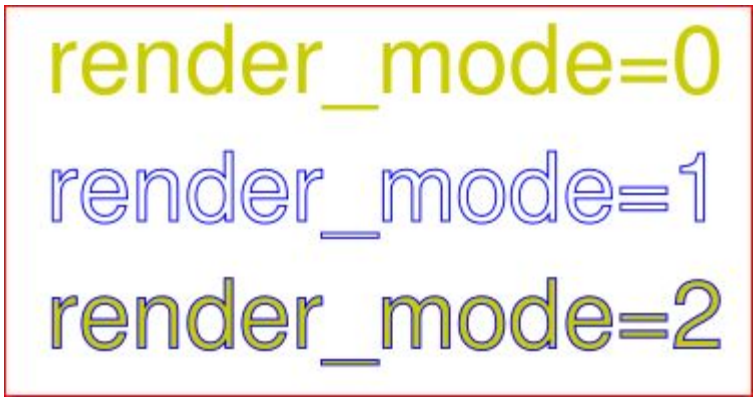
render_mode (*int*)

New in version 1.14.9: Integer in `range(8)` which controls the text appearance (*`Shape.insertText()`* and *`Shape.insertTextbox()`*). See page 398 in *Adobe PDF Reference 1.7*. New in v1.14.9. These methods now also differentiate between fill and stroke colors.

- For default 0, only the text fill color is used to paint the text. For backward compatibility, using the `color` parameter instead also works.
- For render mode 1, only the border of each glyph (i.e. text character) is drawn with a thickness as set in argument `border_width`. The color chosen in the `color` argument is taken for this, the `fill` parameter is ignored.
- For render mode 2, the glyphs are filled and stroked, using both color parameters and the specified border width. You can use this value to simulate **bold text** without using another font: choose the same value for `fill` and `color` and an appropriate value for `border_width`.
- For render mode 3, the glyphs are neither stroked nor filled: the text becomes invisible.

Note: This version 1.14.0 of the base library MuPDF contains a bug: text with render modes 2 and 6 is returned twice and must be dealt with in your script. A fix can be expected with the next MuPDF version.

The following examples use border_width=0.3, together with a fontsize of 15. Stroke color is blue and fill color is some yellow.



---

overlay (*bool*)

Causes the item to appear in foreground (default) or background.

---

morph (*sequence*)

Causes "morphing" of either a shape, created by the `draw*()` methods, or the text inserted by page methods `insertTextbox()` / `insertText()`. If not `None`, it must be a pair `(pivot, matrix)`, where `pivot` is a *Point* and `matrix` is a *Matrix*. The matrix can be anything except translations, i.e. `matrix.e == matrix.f == 0` must be true. The point is used as a pivotal point for the matrix operation. For example, if `matrix` is a rotation or scaling operation, then `pivot` is its center. Similarly, if `matrix` is a left-right or up-down flip, then the mirroring axis will be the vertical, respectively horizontal line going through `pivot`, etc.

Note: Several methods contain checks whether the to be inserted items will actually fit into the page (like *`Shape.insertText()`*, or *`Shape.drawRect()`*). For the result of a morphing operation there is however no such guaranty: this is entirely the rpogrammer's responsibility.

---

lineCap (deprecated: "roundCap") (*int*)

Controls the look of line ends. The default value 0 lets each line end at exactly the given coordinate in a sharp edge. A value of 1 adds a semi-circle to the ends, whose center is the end point and whose diameter is the line width. Value 2 adds a semi-square with an edge length of line width and a center of the line end.

Changed in version 1.14.15.

---

lineJoin (*int*)

New in version 1.14.15: Controls the way how line connections look like. This may be either as a sharp edge (0), a rounded join (1), or a cut-off edge (2, "butt").

---

closePath (*bool*)

Causes the end point of a drawing to be automatically connected with the starting point (by a straight line).

5.17 TextPage

This class represents text and images shown on a document page. All MuPDF document types are supported.

The usual ways to create a textpage are *DisplayList.getTextPage()* and *Page.getTextPage()*. Because there is a limited set of methods in this class, there exist wrappers in the *Page* class, which incorporate creating an intermediate text page and then invoke one of the following methods. The last column of this table shows these corresponding *Page* methods.

For a description of what this class is all about, see Appendix 2.

Method	Description	*Page* wrapper
extractText()	extract plain text	`getText("text")`
extractTEXT()	synonym of previous	`getText("text")`
extractBLOCKS()	plain text grouped in blocks	`getText("blocks")`
extractWORDS()	all words with their bbox	`getText("words")`
extractHTML()	page content in HTML format	`getText("html")`
extractJSON()	page content in JSON format	`getText("json")`
extractXHTML()	page content in XHTML format	`getText("xhtml")`
extractXML()	page text in XML format	`getText("xml")`
extractDICT()	page content in *dict* format	`getText("dict")`
extractRAWDICT()	page content in *dict* format	`getText("rawdict")`
search()	Search for a string in the page	`searchFor()`

Class API

`class TextPage`

`extractText()`

`extractTEXT()`
Return a string of the page's complete text. The text is UTF-8 unicode and in the same sequence as specified at the time of document creation.

Return type str

`extractBLOCKS()`
Textpage content as a list of text lines grouped by block. Each list items looks like this:

```
(x0, y0, x1, y1, "lines in blocks", block_type, block_no)
```

The first four entries are the block's bbox coordinates, `block_type` is 1 for an image block, 0 for text. `block_no` is the block sequence number.

For an image block, its bbox and a text line with image meta information is included – not the image data itself.

This is a high-speed method with enough information to rebuild a desired text sequence.

Return type list

`extractWORDS()`
Textpage content as a list of single words with bbox information. An item of this list looks like this:

```
(x0, y0, x1, y1, "word", block_no, line_no, word_no)
```

Everything wrapped in spaces is treated as a *"word"* with this method.

This is a high-speed method which e.g. allows extracting text from within a given rectangle.

Return type list

`extractHTML()`
Textpage content in HTML format. This version contains complete formatting and positioning information. Images are included (encoded as base64 strings). You need an HTML package to interpret the output in Python. Your internet browser should be able to adequately display this information, but see *Controlling Quality of HTML Output*.

Return type str

`extractDICT()`
Textpage content as a Python dictionary. Provides same information detail as HTML. See below for the structure.

Return type dict

`extractJSON()`
Textpage content in JSON format. Created by `json.dumps(TextPage.extractDICT())`. It is included for backlevel compatibility. You will probably use this method ever only for outputting the result in some file. The method detects binary image data, like `bytearray` and `bytes` (Python 3 only) and converts them to base64 encoded strings on JSON output.

Return type str

`extractXHTML()`
Textpage content in XHTML format. Text information detail is comparable with *extractTEXT()*, but also contains images (base64 encoded). This method makes no attempt to re-create the original visual appearance.

Return type str

`extractXML()`
Textpage content in XML format. This contains complete formatting information about every single character on the page: font, size, line, paragraph, location, color, etc. Contains no images. You probably need an XML package to interpret the output in Python.

Return type str

`extractRAWDICT()`
Textpage content as a Python dictionary – technically similar to *extractDICT()*, and it contains that information as a subset (including any images). It provides additional detail down to each character, which makes using XML obsolete in many cases. See below for the structure.

Return type dict

`search(`*string*, *hit_max = 16*, *quads = False*`)`
Search for `string` and return a list of found locations.

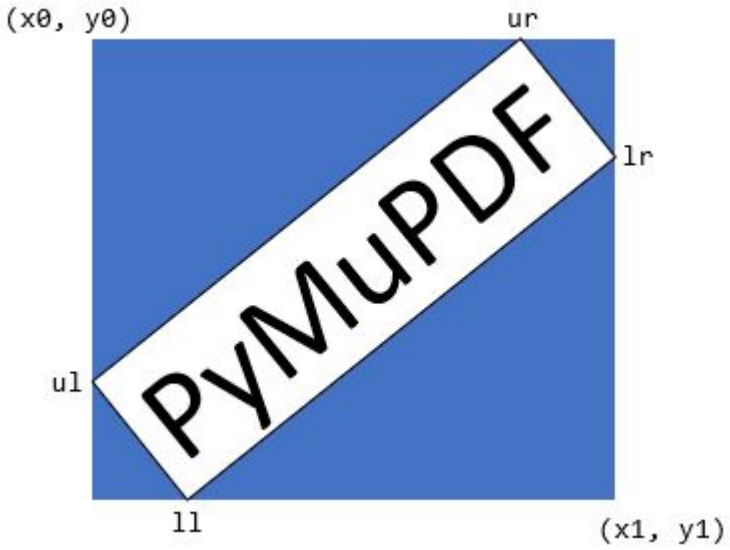
Parameters

- `string` (*str*) – the string to search for. Upper / lower cases will all match.
- `hit_max` (*int*) – maximum number of returned hits (default 16).
- `quads` (*bool*) – return quadrilaterals instead of rectangles.

Return type list

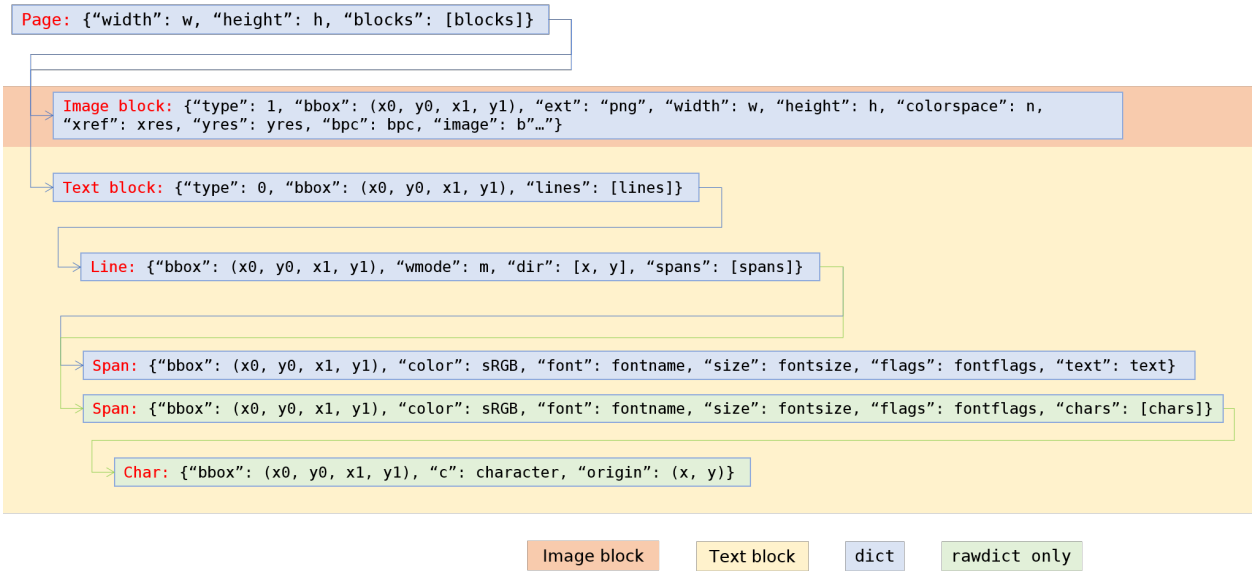
Returns a list of *Rect* or *Quad* objects, each surrounding a found `string` occurrence. The search string may contain spaces, it may therefore happen, that its parts are located on different lines. In this case, more than one rectangle (resp. quadrilateral) are returned. The method does **not support hyphenation**, so it will not find "method" when searching for "method".

Example: If the search for string "pymupdf" contains a hit like shown, then the corresponding entry will either be the blue rectangle, or, if `quads` was specified, `Quad(ul, ur, ll, lr)`.



5.17.1 Dictionary Structure of `extractDICT()` and `extractRAWDICT()`

Visual overview: the TextPage dictionary structure



5.17.1.1 Page Dictionary

Key	Value
width	page width in pixels *(float)*
height	page height in pixels *(float)*
blocks	*list* of block dictionaries

5.17.1.2 Block Dictionaries

Blocks come in two different formats: **image blocks** and **text blocks**.

Image block:

Key	Value
type	1 = image *(int)*
bbox	block / image rectangle, formatted as `tuple(fitz.Rect)`
ext	image type *(str)*, as file extension, see below
width	original image width *(int)*
height	original image height *(int)*
colorspace	colorspace.n *(int)*
xres	resolution in x-direction *(int)*
yres	resolution in y-direction *(int)*
bpc	bits per component *(int)*
image	image content *(bytes or bytearray)*

Possible values of key "ext" are "bmp", "gif", "jpeg", "jpx" (JPEG 2000), "jxr" (JPEG XR), "png", "pnm", and "tiff".

Note:

1. In some error situations, all of the above values may be zero or empty. So, please be prepared to digest items like:

```
{"type": 1, "bbox": (0.0, 0.0, 0.0, 0.0), ..., "image": b""}
```

2. *TextPage* and corresponding method `Page.getText`() are **available for all document types**. Only for PDF documents, methods *Document.getPageImageList()* / :meth'Page.getImageList' offer some overlapping functionality as far as image lists are concerned. But both lists **may or may not** contain the same items. Any differences are most probably caused by one of the following:

   - "Inline" images (see page 352 of the *Adobe PDF Reference 1.7*) of a PDF page are contained in a textpage, but **not in** *Page.getImageList()* .
   - Image blocks in a textpage are generated for **every** image location – whether or not there are any duplicates. This is in contrast to *Page.getImageList()* , which will contain each image only once.
   - Images mentioned in the page's *object* definition will **always** appear in *Page.getImageList()*[86]. But it may happen, that there is no "display" command in the page's

[86] Image specifications for a PDF page are done in the page's sub-dictionary `/Resources`. Being a text format specification, PDF does not prevent one from having arbitrary image entries in this dictionary – whether actually in use by the page or not. On top of this, resource dictionaries can be **inherited** from the page's parent object – like a node of the PDF's *pagetree* or the *catalog* object. So the PDF creator may e.g. define one file level `/Resources` naming all images and fonts ever used by any page. In this case, *Page.getImageList()* and *Page.getFontList()* will always return the same lists for all pages.

contents (erroneously or on purpose). In this case the image will **not appear** in the textpage.

Text block:

Key	Value
type	0 = text *(int)*
bbox	block rectangle, formatted as `tuple(fitz.Rect)`
lines	*list* of text line dictionaries

5.17.1.3 Line Dictionary

Key	Value
bbox	line rectangle, formatted as `tuple(fitz.Rect)`
wmode	writing mode *(int)*: 0 = horizontal, 1 = vertical
dir	writing direction *(list of floats)*: `[x, y]`
spans	*list* of span dictionaries

The value of key `"dir"` is a **unit vetor** and should be interpreted as follows:

- `x`: positive = "left-right", negative = "right-left", 0 = neither
- `y`: positive = "top-bottom", negative = "bottom-top", 0 = neither

The values indicate the "relative writing speed" in each direction, such that $x^2 + y^2 = 1$. In other words `dir = [cos(beta), sin(beta)]`, where `beta` is the writing angle relative to the horizontal.

5.17.1.4 Span Dictionary

Spans contain the actual text. A line contains **more than one span only**, if it contains text with different font properties.

Changed in version 1.14.17: Spans now also have a `bbox` key (again).

Key	Value
bbox	span rectangle, formatted as `tuple(fitz.Rect)`
font	font name *(str)*
size	font size *(float)*
flags	font characteristics *(int)*
color	text color in sRGB format *(int)*
text	(only for `extractDICT()`) text *(str)*
chars	(only for `extractRAWDICT()`) *list* of character dictionaries

New in version 1.16.0: `"color"` is the text color encoded in sRGB format, e.g. 0xFF0000 for red.

`"flags"` is an integer, encoding bools of font properties:

- bit 0: superscripted (2^0)
- bit 1: italic (2^1)
- bit 2: serifed (2^2)
- bit 3: monospaced (2^3)
- bit 4: bold (2^4)

Test these characteristics like so:

```
>>> if flags & 2**1: print("italic")
>>> # etc.
```

5.17.1.5 Character Dictionary for `extractRAWDICT()`

We are currently providing the bbox in *rect_like* format. In a future version, we might change that to *quad_like*. This image shows the relationship between items in the following table:

Key	Value
origin	*tuple* coordinates of the character's bottom left point
bbox	character rectangle, formatted as `tuple(fitz.Rect)`
c	the character (unicode)

5.18 Tools

This class is a collection of utility methods and attributes, mainly around memory management. To simplify and speed up its use, it is automatically instantiated under the name `TOOLS` when PyMuPDF is imported.

Method / Attribute	Description
Tools.gen_id()	generate a unique identifyer
Tools.store_shrink()	shrink the storables cache[88]
Tools.mupdf_warnings()	return the accumulated MuPDF warnings
Tools.reset_mupdf_warnings()	empty MuPDF messages on STDOUT
Tools.fitz_config	configuration settings of PyMuPDF
Tools.store_maxsize	maximum storables cache size
Tools.store_size	current storables cache size

Class API

`class Tools`

`gen_id()`
A convenience method returning a unique positive integer which will increase by 1 on every invocation. Example usages include creating unique keys in databases - its creation should be faster than using timestamps by an order of magnitude.

[88] This memory area is internally used by MuPDF, and it serves as a cache for objects that have already been read and interpreted, thus improving performance. The most bulky object types are images and also fonts. When an application starts up the MuPDF library (in our case this happens as part of `import fitz`), it must specify a maximum size for this area. PyMuPDF's uses the default value (256 MB) to limit memory consumption. Use the methods here to control or investigate store usage. For example: even after a document has been closed and all related objects have been deleted, the store usage may still not drop down to zero. So you might want to enforce that before opening another document.

Note: MuPDF has dropped support for this in v1.14.0, so we have re-implemented a similar function with the following differences:

- It is not part of MuPDF's global context and not threadsafe (because we do not support threads in PyMuPDF yet).
- It is implemented as `int`. This means that the maximum number is `sys.maxsize`. Should this number ever be exceeded, the counter is reset to 1.

Return type int

Returns a unique positive integer.

`store_shrink`(*percent*)

Reduce the storables cache by a percentage of its current size.

Parameters `percent` (*int*) – the percentage of current size to free. If 100+ the store will be emptied, if zero, nothing will happen. MuPDF's caching strategy is "least recently used", so low-usage elements get deleted first.

Return type int

Returns the new current store size. Depending on the situation, the size reduction may be larger than the requested percentage.

`reset_mupdf_warnings`()

New in version 1.16.0: Empty MuPDF warnings message buffer.

`mupdf_warnings`(*reset=True*)

New in version 1.16.0: Return all stored MuPDF messages as a string with interspersed `\n`.

Parameters `reset` (*bool*) – New in version 1.16.7: whether to automatically empty the store.

`fitz_config`

A dictionary containing the actual values used for configuring PyMuPDF and MuPDF. Also refer to the installation chapter. This is an overview of the keys, each of which describes the status of a support aspect.

Key	Support included for ...
plotter-g	Gray colorspace rendering
plotter-rgb	RGB colorspace rendering
plotter-cmyk	CMYK colorspcae rendering
plotter-n	overprint rendering
pdf	PDF documents
xps	XPS documents
svg	SVG documents
cbz	CBZ documents
img	IMG documents
html	HTML documents
epub	EPUB documents
jpx	JPEG2000 images
js	JavaScript
tofu	all TOFU fonts
tofu-cjk	CJK font subset (China, Japan, Korea)
tofu-cjk-ext	CJK font extensions
tofu-cjk-lang	CJK font language extensions
tofu-emoji	TOFU emoji fonts
tofu-historic	TOFU historic fonts
tofu-symbol	TOFU symbol fonts
tofu-sil	TOFU SIL fonts
icc	ICC profiles
py-memory	using Python memory management[89]
base14	Base-14 fonts (should always be true)

For an explanation of the term "TOFU" see this Wikipedia article[87].:

```
In [1]: import fitz
In [2]: TOOLS.fitz_config
Out[2]:
{'plotter-g': True,
 'plotter-rgb': True,
 'plotter-cmyk': True,
 'plotter-n': True,
 'pdf': True,
 'xps': True,
 'svg': True,
 'cbz': True,
 'img': True,
 'html': True,
 'epub': True,
 'jpx': True,
 'js': True,
 'tofu': False,
 'tofu-cjk': True,
 'tofu-cjk-ext': False,
 'tofu-cjk-lang': False,
```

(continues on next page)

[89] Optionally, all dynamic management of memory can be done using Python C-level calls. MuPDF offers a hook to insert user-preferred memory managers. We are using option this for Python version 3 since PyMuPDF v1.13.19. At the same time, all memory allocation in PyMuPDF itself is also routed to Python (i.e. no more direct `malloc()` calls in the code). We have seen improved memory usage and slightly reduced runtimes with this option set. If you want to change this, you can set `#define JM_MEMORY 0` (uses standard C malloc, or 1 for Python allocation)in file `fitz.i` and then generate PyMuPDF.

[87] https://en.wikipedia.org/wiki/Noto_fonts

(continued from previous page)

```
 'tofu-emoji': False,
 'tofu-historic': False,
 'tofu-symbol': False,
 'tofu-sil': False,
 'icc': False,
 'py-memory': True, # (False if Python 2)
 'base14': True}
```

Return type dict

`store_maxsize`
: Maximum storables cache size in bytes. PyMuPDF is generated with a value of 268‘435‘456 (256 MB, the default value), which you should therefore always see here. If this value is zero, then an “unlimited” growth is permitted.

  Return type int

`store_size`
: Current storables cache size in bytes. This value may change (and will usually increase) with every use of a PyMuPDF function. It will (automatically) decrease only when `Tools.store_maxize` is going to be exceeded: in this case, MuPDF will evict low-usage objects until the value is again in range.

  Return type int

5.18.1 Example Session

```
>>> import fitz
# print the maximum and current cache sizes
>>> fitz.TOOLS.store_maxsize
268435456
>>> fitz.TOOLS.store_size
0
>>> doc = fitz.open("demo1.pdf")
# pixmap creation puts lots of object in cache (text, images, fonts),
# apart from the pixmap itself
>>> pix = doc[0].getPixmap(alpha=False)
>>> fitz.TOOLS.store_size
454519
# release (at least) 50% of the storage
>>> fitz.TOOLS.store_shrink(50)
13471
>>> fitz.TOOLS.store_size
13471
# get a few unique numbers
>>> fitz.TOOLS.gen_id()
1
>>> fitz.TOOLS.gen_id()
2
>>> fitz.TOOLS.gen_id()
3
# close document and see how much cache is still in use
>>> doc.close()
>>> fitz.TOOLS.store_size
```

(continues on next page)

(continued from previous page)

```
0
>>>
```

5.19 Widget

This class represents a PDF Form field, also called "widget". Fields are a special case of annotations, which allow users with limited permissions to enter information in a PDF. This is primarily used for filling out forms.

Like annotations, widgets live on PDF pages. Similar to annotations, the first widget on a page is accessible via *Page.firstWidget* and subsequent widgets can be accessed via the *Widget.next* property.

Changed in version 1.16.0: Widgets are no longer mixed with annotations. *Page.firstAnnot* and *Annot.next()* will deliver non-widget annotations exclusively, and be `None` if only form fields exist on a page. Vice versa, *Page.firstWidget* and *Widget.next()* will only show widgets.

Class API

```
class Widget
```

`next`
Point to the next form field on the page.

`update()`
After any changes to a widget, this method **must be used** to store them in the PDF.

`border_color`
A list of up to 4 floats defining the field's border. Default value is `None` which causes border style and border width to be ignored.

`border_style`
A string defining the line style of the field's border. See *Annot.border*. Default is "s" ("Solid") – a continuous line. Only the first character (upper or lower case) will be regarded when creating a widget.

`border_width`
A float defining the width of the border line. Default is 1.

`border_dashes`
A list of integers defining the dash properties of the border line. This is only meaningful if `border_style == "D"` and *border_color* is provided.

`choice_values`
Python sequence of strings defining the valid choices of list boxes and combo boxes. For these widget types the property is mandatory. Ignored for other types. The sequence must contain at least two items. When updating the widget, this sequence will always the complete new list of values must be specified.

`field_name`
A mandatory string defining the field's name. No checking for duplicates takes place.

`field_label`
An optional string containing an "alternate" field name. Typically used for any notes, help on field usage, etc. Default is the field name.

`field_value`
The value of the field.

`field_flags`
An integer defining a large amount of proprties of a field. Handle this attribute with care.

`field_type`
A mandatory integer defining the field type. This is a value in the range of 0 to 6. It cannot be changed when updating the widget.

`field_type_string`
A string describing (and derived from) the field type.

`fill_color`
A list of up to 4 floats defining the field's background color.

`button_caption`
The caption string of a button-type field.

`is_signed`
A bool indicating the status of a signature field, else `None`.

`rect`
The rectangle containing the field.

`text_color`
A list of **1, 3 or 4 floats** defining the text color. Default value is black (*[0, 0, 0]*).

`text_font`
A string defining the font to be used. Default and replacement for invalid values is `"Helv"`. For valid font reference names see the table below.

`text_fontsize`
A float defining the text fontsize. Default value is zero, which causes PDF viewer software to dynamically choose a size suitable for the annotation's rectangle and text amount.

`text_maxlen`
An integer defining the maximum number of text characters. PDF viewers will (should) not accept a longer text.

`text_type`
An integer defining acceptable text types (e.g. numeric, date, time, etc.). For reference only for the time being – will be ignored when creating or updating widgets.

`xref`
An integer defining the PDF cross reference number of the widget.

5.19.1 Standard Fonts for Widgets

Widgets use their own resources object `/DR`. A widget resources object must at least contain a `/Font` object. Widget fonts are independent from page fonts. We currently support the 14 PDF base fonts using the following fixed reference names, or any name of an already existing field font. When specifying a text font for new or changed widgets, **either** choose one in the first table column (upper and lower case supported), **or** one of the already existing form fonts. In the latter case, spelling must exactly match.

To find out already existing field fonts, inspect the list *`Document.FormFonts`*.

Reference	Base14 Fontname
CoBI	Courier-BoldOblique
CoBo	Courier-Bold
CoIt	Courier-Oblique
Cour	Courier
HeBI	Helvetica-BoldOblique
HeBo	Helvetica-Bold
HeIt	Helvetica-Oblique
Helv	Helvetica **(default)**
Symb	Symbol
TiBI	Times-BoldItalic
TiBo	Times-Bold
TiIt	Times-Italic
TiRo	Times-Roman
ZaDb	ZapfDingbats

You are generally free to use any font for every widget. However, we recommend using `ZaDb` ("ZapfDingbats") and fontsize 0 for check boxes: typical viewers will put a correctly sized tickmark in the field's rectangle, when it is clicked.

CHAPTER

SIX

OPERATOR ALGEBRA FOR GEOMETRY OBJECTS

Instances of classes *Point*, *IRect*, *Rect* and *Matrix* are collectively also called "geometry" objects.

They all are special cases of Python sequences, see *Using Python Sequences as Arguments in PyMuPDF* for more background.

We have defined operators for these classes that allow dealing with them (almost) like ordinary numbers in terms of addition, subtraction, multiplication, division, and some others.

This chapter is a synopsis of what is possible.

6.1 General Remarks

1. Operators can be either **binary** (i.e. involving two objects) or **unary**.
2. The resulting type of **binary** operations is either a **new object of the left operand's class** or a bool.
3. The result of **unary** operations is either a **new object** of the same class, a bool or a float.
4. The binary operators `+`, `-`, `*`, `/` are defined for all classes. They *roughly* do what you would expect – **except, that the second operand ...**
   - may always be a number which then performs the operation on every component of the first one,
   - may always be a numeric sequence of the same length (2, 4 or 6) – we call such sequences *point_like*, *rect_like* or *matrix_like*, respectively.
5. Rectangles support additional binary operations: **intersection** (operator `"&"`), **union** (operator `"|"`) and **containment** checking.
6. Binary operators fully support in-place operations, so expressions like `"a /= b"` are valid if b is numeric or "a_like".

6.2 Unary Operations

Oper.	Result
bool(OBJ)	is false exactly if all components of OBJ are zero
abs(OBJ)	the rectangle area – equal to norm(OBJ) for the other tyes
norm(OBJ)	square root of the component squares (Euclidean norm)
+OBJ	new copy of OBJ
-OBJ	new copy of OBJ with negated components
~m	inverse of *Matrix* "m", or the null matrix if not invertible

6.3 Binary Operations

For every geometry object "a" and every number "b", the operations "a ° b" and "a °= b" are always defined for the operators `+, -, *, /`. The respective operation is simply executed for each component of "a". If the **second operand is not a number**, then the following is defined:

Oper.	Result
a+b, a-b	component-wise execution, "b" must be "a-like".
a*m, a/m	"a" can be a point, rectangle or matrix, but "m" must be *matrix_like*. `"a/m"` is treated as `"a*~m"` (see note below for non-invertible matrices). If "a" is a **point** or a **rectangle**, then `"a.transform(m)"` is executed. If "a" is a matrix, then matrix concatenation takes place.
a&b	**intersection rectangle:** "a" must be a rectangle and "b" *rect_like*. Delivers the **largest rectangle** contained in both operands.
a|b	**union rectangle:** "a" must be a rectangle, and "b" may be *point_like* or *rect_like*. Delivers the **smallest rectangle** containing both operands.
b in a	if "b" is a number, then `"b in tuple(a)"` is returned. If "b" is *point_like* or *rect_like*, then "a" must be a rectangle, and `"a.contains(b)"` is returned.
a == b	`True` if `bool(a-b)` is `False` ("b" may be "a-like").

Note: Please note an important difference to usual arithmetics:

Matrix multiplication is **not commutative**, i.e. in general we have $m * n \neq n * m$ for two matrices. Also, there are non-zero matrices which have no inverse, for example `m = Matrix(1, 0, 1, 0, 1, 0)`. If you try to divide by any of these you will receive a `ZeroDivisionError` exception using operator `"/"`, e.g. for `fitz.Identity / m`. But if you formulate `fitz.Identity * ~m`, the result will be `fitz.Matrix()` (the null matrix).

Admittedly, this represents an inconsistency, and we are considering to remove it. For the time being, you can choose to avoid an exception and check whether ~m is the null matrix, or accept a potential ZeroDivisionError by using `fitz.Identity / m`.

6.4 Some Examples

6.4.1 Manipulation with numbers

For the usual arithmetic operations, numbers are always allowed as second operand. In addition, you can formulate `"x in OBJ"`, where x is a number. It is implemented as `"x in tuple(OBJ)"`.

```
>>> fitz.Rect(1, 2, 3, 4) + 5
fitz.Rect(6.0, 7.0, 8.0, 9.0)
>>> 3 in fitz.Rect(1, 2, 3, 4)
True
>>>
```

The following will create the upper left quarter of a document page rectangle:

```
>>> page.rect
Rect(0.0, 0.0, 595.0, 842.0)
>>> page.rect / 2
Rect(0.0, 0.0, 297.5, 421.0)
>>>
```

The following will deliver the **middle point of a line** connecting two points **p1** and **p2**:

```
>>> p1 = fitz.Point(1, 2)
>>> p2 = fitz.Point(4711, 3141)
>>> mp = p1 + (p2 - p1) / 2
>>> mp
Point(2356.0, 1571.5)
>>>
```

6.4.2 Manipulation with "like" Objects

The second operand of a binary operation can always be "like" the left operand. "Like" in this context means "a sequence of numbers of the same length". With the above examples:

```
>>> p1 + p2
Point(4712.0, 3143.0)
>>> p1 + (4711, 3141)
Point(4712.0, 3143.0)
>>> p1 += (4711, 3141)
>>> p1
Point(4712.0, 3143.0)
>>>
```

To shift a rectangle for 5 pixels to the right, do this:

```
>>> fitz.Rect(100, 100, 200, 200) + (5, 0, 5, 0)  # add 5 to the x coordinates
Rect(105.0, 100.0, 205.0, 200.0)
>>>
```

Points, rectangles and matrices can be *transformed* with matrices. In PyMuPDF, we treat this like a **"multiplication"** (or resp. **"division"**), where the second operand may be "like" a matrix. Division in this context means "multiplication with the inverted matrix".

```
>>> m = fitz.Matrix(1, 2, 3, 4, 5, 6)
>>> n = fitz.Matrix(6, 5, 4, 3, 2, 1)
>>> p = fitz.Point(1, 2)
>>> p * m
Point(12.0, 16.0)
>>> p * (1, 2, 3, 4, 5, 6)
Point(12.0, 16.0)
>>> p / m
Point(2.0, -2.0)
>>> p / (1, 2, 3, 4, 5, 6)
Point(2.0, -2.0)
>>>
>>> m * n  # matrix multiplication
Matrix(14.0, 11.0, 34.0, 27.0, 56.0, 44.0)
>>> m / n  # matrix division
Matrix(2.5, -3.5, 3.5, -4.5, 5.5, -7.5)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> m / m  # result is equal to the Identity matrix
Matrix(1.0, 0.0, 0.0, 1.0, 0.0, 0.0)
>>>
>>> # look at this non-invertible matrix:
>>> m = fitz.Matrix(1, 0, 1, 0, 1, 0)
>>> ~m
Matrix(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
>>> # we try dividing by it in two ways:
>>> p = fitz.Point(1, 2)
>>> p * ~m  # this delivers point (0, 0):
Point(0.0, 0.0)
>>> p / m  # but this is an exception:
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    p / m
  File "... /site-packages/fitz/fitz.py", line 869, in __truediv__
    raise ZeroDivisionError("matrix not invertible")
ZeroDivisionError: matrix not invertible
>>>
```

As a specialty, rectangles support additional binary operations:

- **intersection** – the common area of rectangle-likes, operator "&"
- **inclusion** – enlarge to include a point-like or rect-like, operator "|"
- **containment** check – whether a point-like or rect-like is inside

Here is an example for creating the smallest rectangle enclosing given points:

```
>>> # first define some point-likes
>>> points = []
>>> for i in range(10):
        for j in range(10):
            points.append((i, j))
>>>
>>> # now create a rectangle containing all these 100 points
>>> # start with an empty rectangle
>>> r = fitz.Rect(points[0], points[0])
>>> for p in points[1:]:  # and include remaining points one by one
        r |= p
>>> r  # here is the to be expected result:
Rect(0.0, 0.0, 9.0, 9.0)
>>> (4, 5) in r  # this point-like lies inside the rectangle
True
>>> # and this rect-like is also inside
>>> (4, 4, 5, 5) in r
True
>>>
```

CHAPTER

SEVEN

LOW LEVEL FUNCTIONS AND CLASSES

Contains a number of functions and classes for the experienced user. To be used for special needs or performance requirements.

7.1 Functions

The following are miscellaneous functions on a fairly low-level technical detail.

Some functions provide detail access to PDF structures. Others are stripped-down, high performance versions of functions providing more information.

Yet others are handy, general-purpose utilities.

Function	Short Description
Document.FontInfos	PDF only: information on inserted fonts
Annot._cleanContents()	PDF only: clean the annot's *contents* objects
ConversionHeader()	return header string for `getText` methods
ConversionTrailer()	return trailer string for `getText` methods
Document._delXmlMetadata()	PDF only: remove XML metadata
Document._deleteObject()	PDF only: delete an object
Document._getNewXref()	PDF only: create and return a new *xref* entry
Document._getOLRootNumber()	PDF only: return / create *xref* of `/Outline`
Document._getPDFroot()	PDF only: return the *xref* of the catalog
Document._getPageObjNumber()	PDF only: return *xref* and generation number of a page
Document._getPageXref()	PDF only: same as `_getPageObjNumber()`
Document._getTrailerString()	PDF only: return the PDF file trailer string
Document._getXmlMetadataXref()	PDF only: return XML metadata *xref* number
Document._getXrefLength()	PDF only: return length of *xref* table
Document._getXrefStream()	PDF only: return content of a stream object
Document._getXrefString()	PDF only: return object definition "source"
Document._make_page_map()	PDF only: create a fast-access array of page numbers
Document._updateObject()	PDF only: insert or update a PDF object
Document._updateStream()	PDF only: replace the stream of an object
Document.extractFont()	PDF only: extract embedded font
Document.extractImage()	PDF only: extract embedded image
Document.getCharWidths()	PDF only: return a list of glyph widths of a font
Document.isStream()	PDF only: check whether an *xref* is a stream object
ImageProperties()	return a dictionary of basic image properties
getPDFnow()	return the current timestamp in PDF format

Continued on next page

Table 1 – continued from previous page

Function	Short Description
getPDFstr()	return PDF-compatible string
getTextlength()	return string length for a given font & fontsize
Page._cleanContents()	PDF only: clean the page's *contents* objects
Page._getContents()	PDF only: return a list of content numbers
Page._setContents()	PDF only: set page's *contents* object to specified *xref*
Page.getDisplayList()	create the page's display list
Page.getTextBlocks()	extract text blocks as a Python list
Page.getTextWords()	extract text words as a Python list
Page.run()	run a page through a device
Page._wrapContents()	wrap contents with stacking commands
Page._isWrapped	check whether contents wrapping is present
planishLine()	matrix to map a line to the x-axis
PaperSize()	return width, height for a known paper format
PaperRect()	return rectangle for a known paper format
paperSizes	dictionary of pre-defined paper formats

PaperSize(*s*)

Convenience function to return width and height of a known paper format code. These values are given in pixels for the standard resolution 72 pixels = 1 inch.

Currently defined formats include **'A0'** through **'A10'**, **'B0'** through **'B10'**, **'C0'** through **'C10'**, **'Card-4x6'**, **'Card-5x7'**, **'Commercial'**, **'Executive'**, **'Invoice'**, **'Ledger'**, **'Legal'**, **'Legal-13'**, **'Letter'**, **'Monarch'** and **'Tabloid-Extra'**, each in either portrait or landscape format.

A format name must be supplied as a string (case **in** sensitive), optionally suffixed with "-L" (landscape) or "-P" (portrait). No suffix defaults to portrait.

Parameters `s` (*str*) – any format name from above (upper or lower case), like `"A4"` or `"letter-l"`.

Return type tuple

Returns `(width, height)` of the paper format. For an unknown format `(-1, -1)` is returned. Esamples: `fitz.PaperSize("A4")` returns `(595, 842)` and `fitz.PaperSize("letter-l")` delivers `(792, 612)`.

PaperRect(*s*)

Convenience function to return a *Rect* for a known paper format.

Parameters `s` (*str*) – any format name supported by *PaperSize()*.

Return type *Rect*

Returns `fitz.Rect(0, 0, width, height)` with `width, height=fitz.PaperSize(s)`.

```
>>> import fitz
>>> fitz.PaperRect("letter-l")
fitz.Rect(0.0, 0.0, 792.0, 612.0)
>>>
```

---

planishLine(*p1*, *p2*)

New in version 1.16.2: Return a matrix which maps the line from p1 to p2 to the x-axis such that p1 will become (0,0) and p2 a point with the same distance to (0,0).

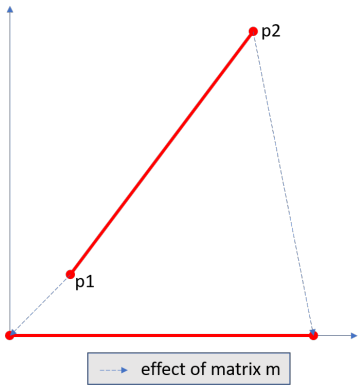
Parameters

- p1 (*point_like*) – starting point of the line.
- p2 (*point_like*) – end point of the line.

Return type *Matrix*

Returns a matrix which combines a rotation and a translation.

```
>>> p1 = fitz.Point(1, 1)
>>> p2 = fitz.Point(4, 5)
>>> abs(p2 - p1)  # distance of points
5.0
>>> m = fitz.planishLine(p1, p2)
>>> p1 * m
Point(0.0, 0.0)
>>> p2 * m
Point(5.0, -5.960464477539063e-08)
>>> # distance of the resulting points
>>> abs(p2 * m - p1 * m)
5.0
```



---

paperSizes

A dictionary of pre-defines paper formats. Used as basis for *PaperSize()*.

---

getPDFnow()

Convenience function to return the current local timestamp in PDF compatible format, e.g. `D:20170501121525-04'00'` for local datetime May 1, 2017, 12:15:25 in a timezone 4 hours westward of the UTC meridian.

Return type str

Returns current local PDF timestamp.

---

getTextlength(*text*, *fontname="helv"*, *fontsize=11*, *encoding=TEXT_ENCODING_LATIN*)

New in version 1.14.7: Calculate the length of text on output with a given **builtin** font, fontsize and encoding.

Parameters

- `text` (*str*) – the text string.
- `fontname` (*str*) – the fontname. Must be one of either the *PDF Base 14 Fonts* or the CJK fonts, identified by their "reserved" fontnames (see table in :meth.'Page.insertFont').
- `fontsize` (*float*) – size of the font.
- `encoding` (*int*) – the encoding to use. Besides 0 = Latin, 1 = Greek and 2 = Cyrillic (Russian) are available. Relevant for Base-14 fonts "Helvetica", "Courier" and "Times" and their variants only. Make sure to use the same value as in the corresponding text insertion.

Return type float

Returns the length in points the string will have (e.g. when used in *Page.insertText()*).

Note: This function will only does the calculation – neither does it insert the font nor write the text.

Warning: If you use this function to determine the required rectangle width for the (*Page* or *Shape*) `insertTextbox` methods, be aware that they calculate on a **by-character level**. Because of rounding effects, this will mostly lead to a slightly larger number: `sum([fitz.getTextlength(c) for c in text]) > fitz.getTextlength(text)`. So either (1) do the same, or (2) use something like `fitz.getTextlength(text + "'")` for your calculation.

`getPDFstr`(*text*)

Make a PDF-compatible string: if the text contains code points `ord(c) > 255`, then it will be converted to UTF-16BE with BOM as a hexadecimal character string enclosed in "<>" brackets like `<feff...>`. Otherwise, it will return the string enclosed in (round) brackets, replacing any characters outside the ASCII range with some special code. Also, every "(", ")" or backslash is escaped with an additional backslash.

Parameters `text` (*str*) – the object to convert

Return type str

Returns PDF-compatible string enclosed in either `()` or <>.

`ImageProperties`(*image*)

Parameters `image` (*bytes|bytearray|BytesIO|file*) – an image either in memory or an **opened** file. A memory resident image maybe any of the formats `bytes`, `bytearray` or `io.BytesIO`.

Returns

a dictionary with the following keys (an empty dictionary for any error):

Key	Value
width	(int) width in pixels
height	(int) height in pixels
colorspace	(int) colorspace.n (e.g. 3 = RGB)
bpc	(int) bits per component (usually 8)
format	(int) image format in `range(15)`
ext	(str) suggested image file extension for the format
size	(int) length of the image in bytes

Example:

```
>>> fitz.ImageProperties(open("img-clip.jpg","rb"))
{'bpc': 8, 'format': 9, 'colorspace': 3, 'height': 325, 'width': 244, 'ext': 'jpeg',
↪ 'size': 14161}
>>>
```

`ConversionHeader(`*"text"*, *filename="UNKNOWN"*`)`

Return the header string required to make a valid document out of page text outputs.

Parameters

- `output` (*str*) – type of document. Use the same as the output parameter of `getText()`.
- `filename` (*str*) – optional arbitrary name to use in output types "json" and "xml".

Return type str

`ConversionTrailer(`*output*`)`

Return the trailer string required to make a valid document out of page text outputs. See *Page.getText()* for an example.

Parameters `output` (*str*) – type of document. Use the same as the output parameter of `getText()`.

Return type str

`Document._deleteObject(`*xref*`)`

PDF only: Delete an object given by its cross reference number.

Parameters `xref` (*int*) – the cross reference number. Must be within the document's valid *xref* range.

Warning: Only use with extreme care: this may make the PDF unreadable.

`Document._delXmlMetadata()`

Delete an object containing XML-based metadata from the PDF. (Py-) MuPDF does not support XML-based metadata. Use this if you want to make sure that the conventional metadata dictionary will be used exclusively. Many thirdparty PDF programs insert their

own metadata in XML format and thus may override what you store in the conventional dictionary. This method deletes any such reference, and the corresponding PDF object will be deleted during next garbage collection of the file.

Document._getTrailerString(*compressed=False*)

New in version 1.14.9: Return the trailer of the PDF (UTF-8), which is usually located at the PDF file's end. If not a PDF or the PDF has no trailer (because of irrecoverable errors), None is returned.

Parameters compressed (*bool*) – New in version 1.14.14: whether to generate a compressed output or one with nice indentations to ease reading (default).

Returns a string with the PDF trailer information. This is the analogous method to *Document._getXrefString()* except that the trailer has no identifying *xref* number. As can be seen here, the trailer object points to other important objects:

```
>>> doc=fitz.open("adobe.pdf")
>>> # compressed output
>>> print(doc._getTrailerString(True))
<</Size 334093/Prev 25807185/XRefStm 186352/Root 333277 0 R/Info 109959 0 R
/ID[(\\227\\366/gx\\016ds\\244\\207\\326\\261\\\\\\305\\376u)
(H\\323\\177\\346\\371pkF\\243\\262\\375\\346\\325\\002)]>>
>>> # non-compressed otput:
>>> print(doc._getTrailerString(False))
<<
   /Size 334093
   /Prev 25807185
   /XRefStm 186352
   /Root 333277 0 R
   /Info 109959 0 R
   /ID [ (\227\366/gx\016ds\244\207\326\261\\\305\376u)␣
↪(H\323\177\346\371pkF\243\262\375\346\325\002) ]
>>
```

Note: MuPDF is capable of recovering from a number of damages a PDF may have. This includes re-generating a trailer, where the end of a file has been lost (e.g. because of incomplete downloads). If however None is returned for a PDF, then the recovery mechanisms were unsuccessful and you should check for any error messages (Document.openErrCode, Document.openErrMsg, Tools.fitz_stderr).

Document._make_page_map()

Create an internal array of page numbers, which significantly speeds up page lookup (*Document.loadPage()*). If this array exists, finding a page object will be up to two times faster. Functions which change the PDF's page layout (copy, delete, move, select pages) will destroy this array again.

Document._getXmlMetadataXref()

Return the XML-based metadata *xref* of the PDF if present – also refer to *Document._delXmlMetadata()*. You can use it to retrieve the content via *Document._getXrefStream()* and then work with it using some XML software.

Return type int

Returns *xref* of PDF file level XML metadata.

Document._getPageObjNumber(*pno*)
or
Document._getPageXref(*pno*)

Return the *xref* and generation number for a given page.

Parameters pno (*int*) – Page number (zero-based).

Return type list

Returns *xref* and generation number of page pno as a list [xref, gen].

Document._getPDFroot()

Return the *xref* of the PDF catalog.

Return type int

Returns *xref* of the PDF catalog – a central *dictionary* pointing to many other PDF information.

Page.run(*dev*, *transform*)
Run a page through a device.

Parameters

- dev (*Device*) – Device, obtained from one of the *Device* constructors.
- transform (*Matrix*) – Transformation to apply to the page. Set it to *Identity* if no transformation is desired.

Page._wrapContents()
Put string pair "q" / "Q" before, resp. after a page's /Contents object(s) to ensure that any "geometry" changes are **local** only.

Use this method as an alternative, minimalistic version of *Page._cleanContents()*. Its advantage is a small footprint in terms of processing time and impact on incremental saves.

Page._isWrapped
Indicate whether *Page._wrapContents()* may be required for object insertions in standard PDF geometry. Please note that this is a quick, basic check only: a value of False may still be a false alarm.

Page.getTextBlocks(*flags=None*)
Deprecated wrapper for *TextPage.extractBLOCKS()*.

Page.getTextWords(*flags=None*)
Deprecated wrapper for *TextPage.extractWORDS()*.

Page.getDisplayList()
Run a page through a list device and return its display list.

Return type *DisplayList*

Returns the display list of the page.

Page._getContents()
Return a list of *xref* numbers of *contents* objects belonging to the page.

Return type list

Returns a list of *xref* integers.

Each page may have zero to many associated contents objects (*stream* s) which contain some operator syntax describing what appears where and how on the page (like text or images, etc. See the *Adobe PDF Reference 1.7*, chapter "Operator Summary", page 985). This function only enumerates the number(s) of such objects. To get the actual stream source, use function *Document._getXrefStream()* with one of the numbers in this list. Use *Document._updateStream()* to replace the content.

Page._setContents(*xref*)
PDF only: Set a given object (identified by its *xref*) as the page's one and only *contents* object. Useful for joining mutiple *contents* objects as in the following snippet:

```
>>> c = b""
>>> xreflist = page._getContents()
>>> for xref in xreflist:
        c += doc._getXrefStream(xref)
>>> doc._updateStream(xreflist[0], c)
>>> page._setContents(xreflist[0])
>>> # doc.save(..., garbage=1) will remove the unused objects
```

Parameters `xref` (*int*) – the cross reference number of a *contents* object. An exception is raised if outside the valid *xref* range or not a stream object.

Page._cleanContents()
Clean and concatenate all *contents* objects associated with this page. "Cleaning" includes syntactical corrections, standardizations and "pretty printing" of the contents stream. Discrepancies between *contents* and *resources* objects will also be corrected. See *Page._getContents()* for more details.

Changed in version 1.16.0: Annotations are no longer implicitely cleaned by this method. Use *Annot._cleanContents()* separately.

Warning: This is a complex function which may generate large amounts of new data and render other data unused. It is **not recommended** using it together with the **incremental save** option. Also note that the resulting singleton new `/Contents` object is **uncompressed**. So you should save to a **new file** using options `"deflate=True, garbage=3"`.

`Annot._cleanContents()`

Clean the *contents* streams associated with the annotation. This is the same type of action which *Page._cleanContents()* performs – just restricted to this annotation.

`Document.getCharWidths(`*xref=0*, *limit=256*`)`

Return a list of character glyphs and their widths for a font that is present in the document. A font must be specified by its PDF cross reference number *xref*. This function is called automatically from *Page.insertText()* and *Page.insertTextbox()*. So you should rarely need to do this yourself.

Parameters

- `xref` (*int*) – cross reference number of a font embedded in the PDF. To find a font *xref*, use e.g. `doc.getPageFontList(pno)` of page number `pno` and take the first entry of one of the returned list entries.
- `limit` (*int*) – limits the number of returned entries. The default of 256 is enforced for all fonts that only support 1-byte characters, so-called "simple fonts" (checked by this method). All *PDF Base 14 Fonts* are simple fonts.

Return type list

Returns a list of `limit` tuples. Each character `c` has an entry `(g, w)` in this list with an index of `ord(c)`. Entry `g` (integer) of the tuple is the glyph id of the character, and float `w` is its normalized width. The actual width for some font-size can be calculated as `w * fontsize`. For simple fonts, the `g` entry can always be safely ignored. In all other cases `g` is the basis for graphically representing `c`.

This function calculates the pixel width of a string called `text`:

```
def pixlen(text, widthlist, fontsize):
try:
    return sum([widthlist[ord(c)] for c in text]) * fontsize
except IndexError:
    m = max([ord(c) for c in text])
    raise ValueError:("max. code point found: %i, increase limit" % m)
```

`Document._getXrefString(`*xref*, *compressed=False*`)`

Return the string ("source code") representing an arbitrary object. For *stream* objects, only the non-stream part is returned. To get the stream data, use *_getXrefStream()*.

Parameters

- `xref` (*int*) – *xref* number.

- `compressed` (*bool*) – New in version 1.14.14: whether to generate a compressed output or one with nice indentations to ease reading or parsing (default).

Return type string

Returns the string defining the object identified by *xref*. Example:

```
>>> doc = fitz.open("Adobe PDF Reference 1-7.pdf")  # the PDF
>>> page = doc[100]  # some page in it
>>> print(doc._getXrefString(page.xref, compressed=True))
<</CropBox[0 0 531 666]/Annots[4795 0 R 4794 0 R 4793 0 R 4792 0 R 4797 0 R 4796 0␣
↪R]
/Parent 109820 0 R/StructParents 941/Contents 229 0 R/Rotate 0/MediaBox[0 0 531 666]
/Resources<</Font<</T1_0 3914 0 R/T1_1 3912 0 R/T1_2 3957 0 R/T1_3 3913 0 R/T1_4␣
↪4576 0 R
/T1_5 3931 0 R/T1_6 3944 0 R>>/ProcSet[/PDF/Text]/ExtGState<</GS0 333283 0 R>>>>
/Type/Page>>
>>> print(doc._getXrefString(page.xref, compressed=False))
<<
   /CropBox [ 0 0 531 666 ]
   /Annots [ 4795 0 R 4794 0 R 4793 0 R 4792 0 R 4797 0 R 4796 0 R ]
   /Parent 109820 0 R
   /StructParents 941
   /Contents 229 0 R
   /Rotate 0
   /MediaBox [ 0 0 531 666 ]
   /Resources <<
      /Font <<
         /T1_0 3914 0 R
         /T1_1 3912 0 R
         /T1_2 3957 0 R
         /T1_3 3913 0 R
         /T1_4 4576 0 R
         /T1_5 3931 0 R
         /T1_6 3944 0 R
      >>
      /ProcSet [ /PDF /Text ]
      /ExtGState <<
         /GS0 333283 0 R
      >>
   >>
   /Type /Page
>>
```

`Document.isStream(xref)`

New in version 1.14.14: PDF only: Check whether the object represented by *xref* is a *stream* type. Return is `False` if not a PDF or if the number is outside the valid xref range.

Parameters `xref` (*int*) – *xref* number.

Returns `True` if the object definition is followed by data wrapped in keyword pair `stream`, `endstream`.

`Document._getNewXref()`

Increase the *xref* by one entry and return that number. This can then be used to insert a new object.

Return type int

Returns the number of the new *xref* entry.

Document._updateObject(*xref*, *obj_str*, *page=None*)

Associate the object identified by string `obj_str` with `xref`, which must already exist. If `xref` pointed to an existing object, this will be replaced with the new object. If a page object is specified, links and other annotations of this page will be reloaded after the object has been updated.

Parameters

- `xref` (*int*) – *xref* number.
- `obj_str` (*str*) – a string containing a valid PDF object definition.
- `page` (*Page*) – a page object. If provided, indicates, that annotations of this page should be refreshed (reloaded) to reflect changes incurred with links and / or annotations.

Return type int

Returns zero if successful, otherwise an exception will be raised.

Document._getXrefLength()

Return length of *xref* table.

Return type int

Returns the number of entries in the *xref* table.

Document._getXrefStream(*xref*)

Return the decompressed stream of the object referenced by `xref`. For non-stream objects `None` is returned.

Parameters `xref` (*int*) – *xref* number.

Return type bytes

Returns the (decompressed) stream of the object.

Document._updateStream(*xref*, *stream*, *new=False*)

Replace the stream of an object identified by `xref`. If the object has no stream, an exception is raised unless `new=True` is used. The function automatically performs a compress operation ("deflate") where beneficial.

Parameters

- `xref` (*int*) – *xref* number.
- `stream` (*bytes*/*bytearray*/*BytesIO*) – the new content of the stream.

  Changed in version 1.14.13: `io.BytesIO` objects are now also supported.
- `new` (*bool*) – whether to force accepting the stream, and thus **turning it into a stream object**.

This method is intended to manipulate streams containing PDF operator syntax (see pp. 985 of the *Adobe PDF Reference 1.7*) as it is the case for e.g. page content streams.

If you update a contents stream, you should use save parameter `clean=True`. This ensures consistency between PDF operator source and the object structure.

Example: Let us assume that you no longer want a certain image appear on a page. This can be achieved by deleting the respective reference in its contents source(s) – and indeed: the image will be gone after reloading the page. But the page's *resources* object would still show the image as being referenced by the page. This save option will clean up any such mismatches.

`Document._getOLRootNumber()`

Return *xref* number of the /Outlines root object (this is **not** the first outline entry!). If this object does not exist, a new one will be created.

Return type int

Returns *xref* number of the **/Outlines** root object.

`Document.extractImage(`*xref=0*`)`

PDF Only: Extract data and meta information of an image stored in the document. The output can directly be used to be stored as an image file, as input for PIL, *Pixmap* creation, etc. This method avoids using pixmaps wherever possible to present the image in its original format (e.g. as JPEG).

Parameters `xref` (*int*) – *xref* of an image object. Must be in `range(1, doc._getXrefLength())`, else an exception is raised. If the object is no image or other errors occur, an empty dictionary is returned and no exception occurs.

Return type dict

Returns

a dictionary with the following keys

- `ext` (*str*) image type (e.g. `'jpeg'`), usable as image file extension
- `smask` (*int*) *xref* number of a stencil (/SMask) image or zero
- `width` (*int*) image width
- `height` (*int*) image height
- `colorspace` (*int*) the image's `pixmap.n` number (indicative only: depends on whether internal pixmaps had to be used). Zero for JPX images.
- `cs-name` (*str*) the image's `colorspace.name`.
- `xres` (*int*) resolution in x direction. Zero for JPX images.
- `yres` (*int*) resolution in y direction. Zero for JPX images.
- `image` (*bytes*) image data, usable as image file content

```
>>> d = doc.extractImage(25)
>>> d
{}
>>> d = doc.extractImage(1373)
>>> d
```

(continues on next page)

(continued from previous page)

```
{'ext': 'png', 'smask': 2934, 'width': 5, 'height': 629, 'colorspace': 3, 'xres':␣
↪96,
'yres': 96, 'cs-name': 'DeviceRGB',
'image': b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x00\x05\ ...'}
>>> imgout = open("image." + d["ext"], "wb")
>>> imgout.write(d["image"])
102
>>> imgout.close()
```

Note: There is a functional overlap with `pix = fitz.Pixmap(doc, xref)`, followed by a `pix.getPNGData()`. Main differences are that extractImage **(1)** does not only deliver PNG image formats, **(2)** is **very** much faster with non-PNG images, **(3)** usually results in much less disk storage for extracted images, **(4)** generates an empty *dict* for non-image xrefs (generates no exception). Look at the following example images within the same PDF.

- xref 1268 is a PNG – Comparable execution time and identical output:

```
In [23]: %timeit pix = fitz.Pixmap(doc, 1268);pix.getPNGData()
10.8 ms ± 52.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
In [24]: len(pix.getPNGData())
Out[24]: 21462

In [25]: %timeit img = doc.extractImage(1268)
10.8 ms ± 86 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
In [26]: len(img["image"])
Out[26]: 21462
```

- xref 1186 is a JPEG – *Document.extractImage()* is **thousands of times faster** and produces a **much smaller** output (2.48 MB vs. 0.35 MB):

```
In [27]: %timeit pix = fitz.Pixmap(doc, 1186);pix.getPNGData()
341 ms ± 2.86 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [28]: len(pix.getPNGData())
Out[28]: 2599433

In [29]: %timeit img = doc.extractImage(1186)
15.7 µs ± 116 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
In [30]: len(img["image"])
Out[30]: 371177
```

`Document.extractFont`(*xref*, *info_only=False*)

PDF Only: Return an embedded font file's data and appropriate file extension. This can be used to store the font as an external file. The method does not throw exceptions (other than via checking for PDF and valid *xref*).

Parameters

- `xref` (*int*) – PDF object number of the font to extract.
- `info_only` (*bool*) – only return font information, not the buffer. To be used for information-only purposes, avoids allocation of large buffer areas.

Return type tuple

Returns

a tuple `(basename, ext, subtype, buffer)`, where `ext` is a 3-byte suggested file extension (*str*), `basename` is the font's name (*str*), `subtype` is the font's type (e.g. "Type1") and `buffer` is a bytes object containing the font file's content (or `b""`). For possible extension values and their meaning see *Font File Extensions*. Return details on error:

- `("", "", "", b"")` – invalid xref or xref is not a (valid) font object.
- `(basename, "n/a", "Type1", b"")` – `basename` is one of the *PDF Base 14 Fonts*, which cannot be extracted.

Example:

```
>>> # store font as an external file
>>> name, ext, buffer = doc.extractFont(4711)
>>> # assuming buffer is not None:
>>> ofile = open(name + "." + ext, "wb")
>>> ofile.write(buffer)
>>> ofile.close()
```

Warning: The basename is returned unchanged from the PDF. So it may contain characters (such as blanks) which may disqualify it as a filename for your operating system. Take appropriate action.

`Document.FontInfos`

Contains following information for any font inserted via *`Page.insertFont()`* in **this** session of PyMuPDF:

- xref *(int)* – XREF number of the `/Type/Font` object.
- info *(dict)* – detail font information with the following keys:
  - name *(str)* – name of the basefont
  - idx *(int)* – index number for multi-font files
  - type *(str)* – font type (like "TrueType", "Type0", etc.)
  - ext *(str)* – extension to be used, when font is extracted to a file (see *Font File Extensions*).
  - glyphs (*list*) – list of glyph numbers and widths (filled by textinsertion methods).

Return type list

7.2 Device

The different format handlers (pdf, xps, etc.) interpret pages to a "device". Devices are the basis for everything that can be done with a page: rendering, text extraction and searching. The device type is determined by the selected construction method.

Class API

`class Device`

__init__(*self*, *object*, *clip*)
Constructor for either a pixel map or a display list device.

Parameters

- `object` (*Pixmap* or *DisplayList*) – either a `Pixmap` or a `DisplayList`.
- `clip` (*IRect*) – An optional *IRect* for `Pixmap` devices to restrict rendering to a certain area of the page. If the complete page is required, specify `None`. For display list devices, this parameter must be omitted.

__init__(*self*, *textpage*, *flags=0*)
Constructor for a text page device.

Parameters

- `textpage` (*TextPage*) – `TextPage` object
- `flags` (*int*) – control the way how text is parsed into the text page. Currently 3 options can be coded into this parameter, see *Preserve Text Flags*. To set these options use something like `flags=0 | TEXT_PRESERVE_LIGATURES | ...`.

Note: In higher level code (*Page.getText()*, *Document.getPageText()*), the following decisions for creating text devices have been implemented: (1) `TEXT_PRESERVE_LIGATURES` and `TEXT_PRESERVE_WHITESPACES` are always set, (2) `TEXT_PRESERVE_IMAGES` is set for JSON and HTML, otherwise off.

7.3 Working together: DisplayList and TextPage

Here are some instructions on how to use these classes together.

In some situations, performance improvements may be achievable, when you fall back to the detail level explained here.

7.3.1 Create a DisplayList

A *DisplayList* represents an interpreted document page. Methods for pixmap creation, text extraction and text search are – behind the curtain – all using the page's display list to perform their tasks. If a page must be rendered several times (e.g. because of changed zoom levels), or if text search and text extraction should both be performed, overhead can be saved, if the display list is created only once and then used for all other tasks.

```
>>> dl = page.getDisplayList()              # create the display list
```

You can also create display lists for many pages "on stack" (in a list), may be during document open, during idling times, or you store it when a page is visited for the first time (e.g. in GUI scripts).

Note, that for everything what follows, only the display list is needed – the corresponding *Page* object could have been deleted.

7.3.2 Generate Pixmap

The following creates a Pixmap from a *DisplayList*. Parameters are the same as for *Page.getPixmap()*.

```
>>> pix = dl.getPixmap()                    # create the page's pixmap
```

The execution time of this statement may be up to 50% shorter than that of `Page.getPixMap()`.

7.3.3 Perform Text Search

With the display list from above, we can also search for text.

For this we need to create a *TextPage*.

```
>>> tp = dl.getTextPage()                   # display list from above
>>> rlist = tp.search("needle")             # look up "needle" locations
>>> for r in rlist:                         # work with the found locations, e.g.
        pix.invertIRect(r.irect)            # invert colors in the rectangles
```

7.3.4 Extract Text

With the same *TextPage* object from above, we can now immediately use any or all of the 5 text extraction methods.

Note: Above, we have created our text page without argument. This leads to a default argument of `3 = fitz.TEXT_PRESERVE_LIGATURES | fitz.TEXT_PRESERVE_WHITESPACE`, IAW images will **not** be extracted – see below.

```
>>> txt  = tp.extractText()                 # plain text format
>>> json = tp.extractJSON()                 # json format
>>> html = tp.extractHTML()                 # HTML format
>>> xml  = tp.extractXML()                  # XML format
>>> xml  = tp.extractXHTML()                # XHTML format
```

7.3.5 Further Performance improvements

7.3.5.1 Pixmap

As explained in the *Page* chapter:

If you do not need transparency set `alpha = 0` when creating pixmaps. This will save 25% memory (if RGB, the most common case) and possibly 5% execution time (depending on the GUI software).

7.3.5.2 TextPage

If you do not need images extracted alongside the text of a page, you can set the following option:

```
>>> flags = fitz.TEXT_PRESERVE_LIGATURES | fitz.TEXT_PRESERVE_WHITESPACE
>>> tp = dl.getTextPage(flags)
```

This will save ca. 25% overall execution time for the HTML, XHTML and JSON text extractions and **hugely** reduce the amount of storage (both, memory and disk space) if the document is graphics oriented.

If you however do need images, use a value of 7 for flags:

```
>>> flags = fitz.TEXT_PRESERVE_LIGATURES | fitz.TEXT_PRESERVE_WHITESPACE | fitz.TEXT_PRESERVE_
↪IMAGES
```

CHAPTER

EIGHT

GLOSSARY

`matrix_like`
: A Python sequence of 6 numbers.

`rect_like`
: A Python sequence of 4 numbers.

`irect_like`
: A Python sequence of 4 integers.

`point_like`
: A Python sequence of 2 numbers.

`quad_like`
: A Python sequence of 4 *`point_like`* items.

`catalog`
: A central PDF *`dictionary`* containing pointers to many other information.

`contents`
: "A **content stream** is a PDF *`stream object`* whose data consists of a sequence of instructions describing the graphical elements to be painted on a page." (*Adobe PDF Reference 1.7* p. 151). For an overview of the mini-language used in these streams see chapter "Operator Summary" on page 985 of the *Adobe PDF Reference 1.7*. A PDF *`page`* can have none to many contents objects. If it has none, the page is empty (but still may show annotations). If it has several, they will be interpreted in sequence as if their instructions had been present in one such object (i.e. like in a concatenated string). It should be noted that there are more stream object types which use the same syntax: e.g. appearance dictionaries associated with annotations and Form XObjects.

`resources`
: A *`dictionary`* containing references to any resources (like images or fonts) required by a PDF *`page`* (required, inheritable, *Adobe PDF Reference 1.7* p. 145) and certain other objects (Form XObjects). This dictionary appears as a sub-dictionary in the object definition under the key `/Resources`. Being an inheritable object type, there may exist "global" resources for all pages or certain subsets of pages.

`dictionary`
: A PDF *`object`* type, which is somewhat comparable to the same-named Python notion: "A dictionary object is an associative table containing pairs of objects, known as the dictionary's entries. The first element of each entry is the key and the second element is the value. The key must be a name (...). The value can be any kind of object, including another dictionary. A dictionary entry whose value is null (...) is equivalent to an absent entry." (*Adobe PDF Reference 1.7* p. 59).

  Dictionaries are the most important *`object`* type in PDF. Here is an example (describing a *`page`*):

```
<<
/Contents 40 0 R                    % value: an indirect object
/Type/Page                          % value: a name object
/MediaBox[0 0 595.32 841.92]        % value: an array object
/Rotate 0                           % value: a number object
/Parent 12 0 R                      % value: an indirect object
/Resources<<                        % value: a dictionary object
    /ExtGState<</R7 26 0 R>>
    /Font<<
         /R8 27 0 R/R10 21 0 R/R12 24 0 R/R14 15 0 R
         /R17 4 0 R/R20 30 0 R/R23 7 0 R /R27 20 0 R
         >>
    /ProcSet[/PDF/Text]             % value: array of two name objects
    >>
/Annots[55 0 R]                     % value: array, one entry (indirect object)
>>
```

`/Contents`, `/Type`, `/MediaBox`, etc. are **keys**, `40 0 R`, `/Page`, `[0 0 595.32 841.92]`, etc. are the respective **values**. The strings << and >> are used to enclose object definitions.

This example also shows the syntax of **nested** dictionary values: `/Resources` has an object as its value, which in turn is a dictionary with keys like `/ExtGState` (with the value `<</R7 26 0 R>>`, which is another dictionary), etc.

`page`
: A PDF page is a *dictionary* object which defines one page in the document, see *Adobe PDF Reference 1.7* p. 145.

`pagetree`
: "The pages of a document are accessed through a structure known as the page tree, which defines the ordering of pages in the document. The tree structure allows PDF consumer applications, using only limited memory, to quickly open a document containing thousands of pages. The tree contains nodes of two types: intermediate nodes, called page tree nodes, and leaf nodes, called page objects." (*Adobe PDF Reference 1.7* p. 143).

While it is possible to list all page references in just one array, PDFs with many pages are often created using *balanced tree* structures ("page trees") for faster access to any single page. In relation to the total number of pages, this can reduce the average page access time by page number from a linear to some logarithmic order of magnitude.

For fast page access, MuPDF can use its own array in memory – independently from what may or may not be present in the document file. This array is indexed by page number and therefore much faster than even the access via a perfectly balanced page tree.

`object`
: Similar to Python, PDF supports the notion *object*, which can come in eight basic types: boolean values, integer and real numbers, strings, names, arrays, dictionaries, streams, and the null object (*Adobe PDF Reference 1.7* p. 51). Objects can be made identifyable by assigning a label. This label is then called *indirect* object. PyMuPDF supports retrieving definitions of indirect objects via their label (the cross reference number) via *Document._getXrefString()*.

`stream`
: A PDF *object* type which is a sequence of bytes, similar to a string. "However, a PDF application can read a stream incrementally, while a string must be read in its entirety. Furthermore, a stream can be of unlimited length, whereas a string is subject to an implementation limit. For this reason, objects with potentially large amounts of data, such as images and page descriptions, are represented as streams." "A stream consists of a *dictionary* followed by zero or more bytes bracketed between the keywords *stream* and *endstream*":

```
nnn 0 obj
<<
   dictionary definition
>>
stream
... zero or more bytes ...
endstream
endobj
```

See *Adobe PDF Reference 1.7* p. 60. PyMuPDF supports retrieving stream content via *Document._getXrefStream()*. Use *Document.isStream()* to determine whether an object is of stream type.

`unitvector`
: A mathematical notion meaning a vector of norm ("length") 1 – usually the Euclidean norm is implied. In PyMuPDF, this term is restricted to *Point* objects, see *Point.unit*.

`xref`
: Abbreviation for cross-reference number: this is an integer unique identification for objects in a PDF. There exists a cross-reference table (which may consist of several separate segments) in each PDF, which stores the relative position of each object for quick lookup. The cross-reference table is one entry longer than the number of existing object: item zero is reserved and must not be used in any way. Many PyMuPDF classes have an `xref` attribute (which is zero for non-PDFs), and one can find out the total number of objects in a PDF via *Document._getXrefLength()*.

CHAPTER

NINE

CONSTANTS AND ENUMERATIONS

Constants and enumerations of MuPDF as implemented by PyMuPDF. Each of the following variables is accessible as `fitz.variable`.

9.1 Constants

`Base14_Fonts`
Predefined Python list of valid *PDF Base 14 Fonts*.

Return type list

`csRGB`
Predefined RGB colorspace `fitz.Colorspace(fitz.CS_RGB)`.

Return type *Colorspace*

`csGRAY`
Predefined GRAY colorspace `fitz.Colorspace(fitz.CS_GRAY)`.

Return type *Colorspace*

`csCMYK`
Predefined CMYK colorspace `fitz.Colorspace(fitz.CS_CMYK)`.

Return type *Colorspace*

`CS_RGB`
1 – Type of *Colorspace* is RGBA

Return type int

`CS_GRAY`
2 – Type of *Colorspace* is GRAY

Return type int

`CS_CMYK`
3 – Type of *Colorspace* is CMYK

Return type int

`VersionBind`
'x.xx.x' – version of PyMuPDF (these bindings)

Return type string

`VersionFitz`
'x.xxx' – version of MuPDF

Return type string

`VersionDate`
ISO timestamp `YYYY-MM-DD HH:MM:SS` when these bindings were built.

Return type string

Note: The docstring of `fitz` contains information of the above which can be retrieved like so: `print(fitz.__doc__)`, and should look like: `PyMuPDF 1.10.0: Python bindings for the MuPDF 1.10 library, built on 2016-11-30 13:09:13`.

`version`
(VersionBind, VersionFitz, timestamp) – combined version information where `timestamp` is the generation point in time formatted as "YYYYMMDDhhmmss".

Return type tuple

9.2 Document Permissions

Code	Permitted Action
PDF_PERM_PRINT	Print the document
PDF_PERM_MODIFY	Modify the document's contents
PDF_PERM_COPY	Copy or otherwise extract text and graphics
PDF_PERM_ANNOTATE	Add or modify text annotations and interactive form fields
PDF_PERM_FORM	Fill in forms and sign the document
PDF_PERM_ACCESSIBILITY	Obsolete, always permitted
PDF_PERM_ASSEMBLE	Insert, rotate, or delete pages, bookmarks, thumbnail images
PDF_PERM_PRINT_HQ	High quality printing

9.3 PDF encryption method codes

Code	Meaning
PDF_ENCRYPT_KEEP	do not change
PDF_ENCRYPT_NONE	remove any encryption
PDF_ENCRYPT_RC4_40	RC4 40 bit
PDF_ENCRYPT_RC4_128	RC4 128 bit
PDF_ENCRYPT_AES_128	*Advanced Encryption Standard* 128 bit
PDF_ENCRYPT_AES_256	*Advanced Encryption Standard* 256 bit
PDF_ENCRYPT_UNKNOWN	unknown

9.4 Font File Extensions

The table show file extensions you should use when extracting fonts from a PDF file.

Ext	Description
ttf	TrueType font
pfa	Postscript for ASCII font (various subtypes)
cff	Type1C font (compressed font equivalent to Type1)
cid	character identifier font (postscript format)
otf	OpenType font
n/a	built-in font (*PDF Base 14 Fonts* or CJK: cannot be extracted)

9.5 Text Alignment

`TEXT_ALIGN_LEFT`
0 – align left.

`TEXT_ALIGN_CENTER`
1 – align center.

`TEXT_ALIGN_RIGHT`
2 – align right.

`TEXT_ALIGN_JUSTIFY`
3 – align justify.

9.6 Preserve Text Flags

Options controlling the amount of data a text device parses into a *TextPage*.

`TEXT_PRESERVE_LIGATURES`
1 – If set, ligatures are passed through to the application in their original form. Otherwise ligatures are expanded into their constituent parts, e.g. the ligature ffi is expanded into three eparate characters f, f and i.

`TEXT_PRESERVE_WHITESPACE`
2 – If set, whitespace is passed through to the application in its original form. Otherwise any type of horizontal whitespace (including horizontal tabs) will be replaced with space characters of variable width.

`TEXT_PRESERVE_IMAGES`
4 – If set, then images will be stored in the structured text structure.

`TEXT_INHIBIT_SPACES`
8 – If set, we will not try to add missing space characters where there are large gaps between characters.

9.7 Link Destination Kinds

Possible values of *`linkDest.kind`* (link destination kind). For details consult *Adobe PDF Reference 1.7*, chapter 8.2 on pp. 581.

`LINK_NONE`
0 – No destination. Indicates a dummy link.

Return type int

LINK_GOTO
1 – Points to a place in this document.

Return type int

LINK_URI
2 – Points to a URI – typically a resource specified with internet syntax.

Return type int

LINK_LAUNCH
3 – Launch (open) another file (of any "executable" type).

Return type int

LINK_GOTOR
5 – Points to a place in another PDF document.

Return type int

9.8 Link Destination Flags

Note: The rightmost byte of this integer is a bit field, so test the truth of these bits with the `&` operator.

LINK_FLAG_L_VALID
1 (bit 0) Top left x value is valid

Return type bool

LINK_FLAG_T_VALID
2 (bit 1) Top left y value is valid

Return type bool

LINK_FLAG_R_VALID
4 (bit 2) Bottom right x value is valid

Return type bool

LINK_FLAG_B_VALID
8 (bit 3) Bottom right y value is valid

Return type bool

LINK_FLAG_FIT_H
16 (bit 4) Horizontal fit

Return type bool

LINK_FLAG_FIT_V
32 (bit 5) Vertical fit

Return type bool

LINK_FLAG_R_IS_ZOOM
64 (bit 6) Bottom right x is a zoom figure

Return type bool

9.9 Annotation Related Constants

See chapter 8.4.5, pp. 615 of the *Adobe PDF Reference 1.7* for more details.

Annotation Types:

```
PDF_ANNOT_TEXT 0
PDF_ANNOT_LINK 1
PDF_ANNOT_FREETEXT 2
PDF_ANNOT_LINE 3
PDF_ANNOT_SQUARE 4
PDF_ANNOT_CIRCLE 5
PDF_ANNOT_POLYGON 6
PDF_ANNOT_POLYLINE 7
PDF_ANNOT_HIGHLIGHT 8
PDF_ANNOT_UNDERLINE 9
PDF_ANNOT_SQUIGGLY 10
PDF_ANNOT_STRIKEOUT 11
PDF_ANNOT_REDACT 12
PDF_ANNOT_STAMP 13
PDF_ANNOT_CARET 14
PDF_ANNOT_INK 15
PDF_ANNOT_POPUP 16
PDF_ANNOT_FILEATTACHMENT 17
PDF_ANNOT_SOUND 18
PDF_ANNOT_MOVIE 19
PDF_ANNOT_WIDGET 20
PDF_ANNOT_SCREEN 21
PDF_ANNOT_PRINTERMARK 22
PDF_ANNOT_TRAPNET 23
PDF_ANNOT_WATERMARK 24
PDF_ANNOT_3D 25
```

Annotation Flag Bits:

```
PDF_ANNOT_IS_Invisible 1 << (1-1)
PDF_ANNOT_IS_Hidden 1 << (2-1)
PDF_ANNOT_IS_Print 1 << (3-1)
PDF_ANNOT_IS_NoZoom 1 << (4-1)
PDF_ANNOT_IS_NoRotate 1 << (5-1)
PDF_ANNOT_IS_NoView 1 << (6-1)
PDF_ANNOT_IS_ReadOnly 1 << (7-1)
PDF_ANNOT_IS_Locked 1 << (8-1)
PDF_ANNOT_IS_ToggleNoView 1 << (9-1)
PDF_ANNOT_IS_LockedContents 1 << (10-1)
```

Annotation Line Ending Styles:

```
PDF_ANNOT_LE_NONE 0
PDF_ANNOT_LE_SQUARE 1
PDF_ANNOT_LE_CIRCLE 2
PDF_ANNOT_LE_DIAMOND 3
PDF_ANNOT_LE_OPEN_ARROW 4
PDF_ANNOT_LE_CLOSED_ARROW 5
PDF_ANNOT_LE_BUTT 6
PDF_ANNOT_LE_R_OPEN_ARROW 7
PDF_ANNOT_LE_R_CLOSED_ARROW 8
PDF_ANNOT_LE_SLASH 9
```

9.10 Widget Constants

Widget types (`field_type`):

```
PDF_WIDGET_TYPE_UNKNOWN 0
PDF_WIDGET_TYPE_BUTTON 1
PDF_WIDGET_TYPE_CHECKBOX 2
PDF_WIDGET_TYPE_COMBOBOX 3
PDF_WIDGET_TYPE_LISTBOX 4
PDF_WIDGET_TYPE_RADIOBUTTON 5
PDF_WIDGET_TYPE_SIGNATURE 6
PDF_WIDGET_TYPE_TEXT 7
```

Text Widget Subtypes (`text_format`):

```
PDF_WIDGET_TX_FORMAT_NONE 0
PDF_WIDGET_TX_FORMAT_NUMBER 1
PDF_WIDGET_TX_FORMAT_SPECIAL 2
PDF_WIDGET_TX_FORMAT_DATE 3
PDF_WIDGET_TX_FORMAT_TIME 4
```

9.10.1 Widget flags (`field_flags`)

Common to all field types:

```
PDF_FIELD_IS_READ_ONLY 1
PDF_FIELD_IS_REQUIRED 1 << 1
PDF_FIELD_IS_NO_EXPORT 1 << 2
```

Text widgets:

```
PDF_TX_FIELD_IS_MULTILINE  1 << 12
PDF_TX_FIELD_IS_PASSWORD  1 << 13
PDF_TX_FIELD_IS_FILE_SELECT  1 << 20
PDF_TX_FIELD_IS_DO_NOT_SPELL_CHECK  1 << 22
PDF_TX_FIELD_IS_DO_NOT_SCROLL  1 << 23
PDF_TX_FIELD_IS_COMB  1 << 24
PDF_TX_FIELD_IS_RICH_TEXT  1 << 25
```

Button widgets:

```
PDF_BTN_FIELD_IS_NO_TOGGLE_TO_OFF  1 << 14
PDF_BTN_FIELD_IS_RADIO  1 << 15
PDF_BTN_FIELD_IS_PUSHBUTTON  1 << 16
PDF_BTN_FIELD_IS_RADIOS_IN_UNISON  1 << 25
```

Choice widgets:

```
PDF_CH_FIELD_IS_COMBO  1 << 17
PDF_CH_FIELD_IS_EDIT  1 << 18
PDF_CH_FIELD_IS_SORT  1 << 19
PDF_CH_FIELD_IS_MULTI_SELECT  1 << 21
PDF_CH_FIELD_IS_DO_NOT_SPELL_CHECK  1 << 22
PDF_CH_FIELD_IS_COMMIT_ON_SEL_CHANGE  1 << 26
```

9.11 Stamp Annotation Icons

MuPDF has defined the following icons for **rubber stamp** annotations:

```
STAMP_Approved 0
STAMP_AsIs 1
STAMP_Confidential 2
STAMP_Departmental 3
STAMP_Experimental 4
STAMP_Expired 5
STAMP_Final 6
STAMP_ForComment 7
STAMP_ForPublicRelease 8
STAMP_NotApproved 9
STAMP_NotForPublicRelease 10
STAMP_Sold 11
STAMP_TopSecret 12
STAMP_Draft 13
```

CHAPTER

TEN

COLOR DATABASE

Since the introduction of methods involving colors (like *Page.drawCircle()*), a requirement may be to have access to predefined colors.

The fabulous GUI package wxPython[90] has a database of over 540 predefined RGB colors, which are given more or less memorizable names. Among them are not only standard names like "green" or "blue", but also "turquoise", "skyblue", and 100 (not only 50 ...) shades of "gray", etc.

We have taken the liberty to copy this database (a list of tuples) modified into PyMuPDF and make its colors available as PDF compatible float triples: for wxPython's `("WHITE", 255, 255, 255)` we return `(1, 1, 1)`, which can be directly used in `color` and `fill` parameters. We also accept any mixed case of "wHiTe" to find a color.

10.1 Function `getColor()`

As the color database may not be needed very often, one additional import statement seems acceptable to get access to it:

```
>>> # "getColor" is the only method you really need
>>> from fitz.utils import getColor
>>> getColor("aliceblue")
(0.9411764705882353, 0.9725490196078431, 1.0)
>>> #
>>> # to get a list of all existing names
>>> from fitz.utils import getColorList
>>> cl = getColorList()
>>> cl
['ALICEBLUE', 'ANTIQUEWHITE', 'ANTIQUEWHITE1', 'ANTIQUEWHITE2', 'ANTIQUEWHITE3',
'ANTIQUEWHITE4', 'AQUAMARINE', 'AQUAMARINE1'] ...
>>> #
>>> # to see the full integer color coding
>>> from fitz.utils import getColorInfoList
>>> il = getColorInfoList()
>>> il
[('ALICEBLUE', 240, 248, 255), ('ANTIQUEWHITE', 250, 235, 215),
('ANTIQUEWHITE1', 255, 239, 219), ('ANTIQUEWHITE2', 238, 223, 204),
('ANTIQUEWHITE3', 205, 192, 176), ('ANTIQUEWHITE4', 139, 131, 120),
('AQUAMARINE', 127, 255, 212), ('AQUAMARINE1', 127, 255, 212)] ...
```

[90] https://wxpython.org/

10.2 Printing the Color Database

If you want to actually see how the many available colors look like, use scripts colordbRGB.py[91] or colordbHSV.py[92] in the examples directory. They create PDFs (already existing in the same directory) with all these colors. Their only difference is sorting order: one takes the RGB values, the other one the Hue-Saturation-Values as sort criteria. This is a screen print of what these files look like.

[91] https://github.com/pymupdf/PyMuPDF/blob/master/examples/colordbRGB.py
[92] https://github.com/pymupdf/PyMuPDF/blob/master/examples/colordbHSV.py

CHAPTER

ELEVEN

APPENDIX 1: PERFORMANCE

We have tried to get an impression on PyMuPDF's performance. While we know this is very hard and a fair comparison is almost impossible, we feel that we at least should provide some quantitative information to justify our bold comments on MuPDF's **top performance**.

Following are three sections that deal with different aspects of performance:

- document parsing
- text extraction
- image rendering

In each section, the same fixed set of PDF files is being processed by a set of tools. The set of tools varies – for reasons we will explain in the section.

Here is the list of files we are using. Each file name is accompanied by further information: **size** in bytes, number of **pages**, number of bookmarks (**toc** entries), number of **links**, **text** size as a percentage of file size, **KB** per page, PDF **version** and remarks. **text %** and **KB index** are indicators for whether a file is text or graphics oriented.

name	size	pages	toc size	links	text %	KB index	version	remarks
Adobe.pdf	32.472.771	1.310	794	32.096	8,0%	24	PDF 1.6	linearized, text oriented, **many** links / bookmarks
Evolution.pdf	13.497.490	75	15	118	1,1%	176	PDF 1.4	graphics oriented
PyMuPDF.pdf	479.011	47	60	491	13,2%	10	PDF 1.4	text oriented, many links
sdw_2015_01.pdf	14.668.972	100	36	0	2,5%	143	PDF 1.3	graphics oriented
sdw_2015_02.pdf	13.295.864	100	38	0	2,7%	130	PDF 1.4	graphics oriented
sdw_2015_03.pdf	21.224.417	108	35	0	1,9%	192	PDF 1.4	graphics oriented
sdw_2015_04.pdf	15.242.911	108	37	0	2,7%	138	PDF 1.3	graphics oriented
sdw_2015_05.pdf	16.495.887	108	43	0	2,4%	149	PDF 1.4	graphics oriented
sdw_2015_06.pdf	23.447.046	100	38	0	1,6%	229	PDF 1.4	graphics oriented
sdw_2015_07.pdf	14.106.982	100	38	2	2,6%	138	PDF 1.4	graphics oriented
sdw_2015_08.pdf	12.321.995	100	37	0	3,0%	120	PDF 1.4	graphics oriented
sdw_2015_09.pdf	23.409.625	100	37	0	1,5%	229	PDF 1.4	graphics oriented
sdw_2015_10.pdf	18.706.394	100	24	0	2,0%	183	PDF 1.5	graphics oriented
sdw_2015_11.pdf	25.624.266	100	20	0	1,5%	250	PDF 1.4	graphics oriented
sdw_2015_12.pdf	19.111.666	108	36	0	2,1%	173	PDF 1.4	graphics oriented

Decimal point and comma follow European convention

E.g. `Adobe.pdf` and `PyMuPDF.pdf` are clearly text oriented, all other files contain many more images.

11.1 Part 1: Parsing

How fast is a PDF file read and its content parsed for further processing? The sheer parsing performance cannot directly be compared, because batch utilities always execute a requested task completely, in one

go, front to end. `pdfrw` too, has a `lazy` strategy for parsing, meaning it only parses those parts of a document that are required in any moment.

To yet find an answer to the question, we therefore measure the time to copy a PDF file to an output file with each tool, and doing nothing else.

These were the tools

All tools are either platform independent, or at least can run both, on Windows and Unix / Linux (pdftk).

Poppler is missing here, because it specifically is a Linux tool set, although we know there exist Windows ports (created with considerable effort apparently). Technically, it is a C/C++ library, for which a Python binding exists – in so far somewhat comparable to PyMuPDF. But Poppler in contrast is tightly coupled to **Qt** and **Cairo**. We may still include it in future, when a more handy Windows installation is available. We have seen however some analysis[93], that hints at a much lower performance than MuPDF. Our comparison of text extraction speeds also show a much lower performance of Poppler's PDF code base **Xpdf**.

Image rendering of MuPDF also is about three times faster than the one of Xpdf when comparing the command line tools `mudraw` of MuPDF and `pdftopng` of Xpdf – see part 3 of this chapter.

Tool	Description
PyMuPDF	tool of this manual, appearing as "fitz" in reports
pdfrw	a pure Python tool, is being used by rst2pdf, has interface to ReportLab
PyPDF2	a pure Python tool with a very complete function set
pdftk	a command line utility with numerous functions

This is how each of the tools was used:

PyMuPDF:

```
doc = fitz.open("input.pdf")
doc.save("output.pdf")
```

pdfrw:

```
doc = PdfReader("input.pdf")
writer = PdfWriter()
writer.trailer = doc
writer.write("output.pdf")
```

PyPDF2:

```
pdfmerge = PyPDF2.PdfFileMerger()
pdfmerge.append("input.pdf")
pdfmerge.write("output.pdf")
pdfmerge.close()
```

pdftk:

```
pdftk input.pdf output output.pdf
```

Observations

These are our run time findings (in **seconds**, please note the European number convention: meaning of decimal point and comma is reversed):

[93] http://hzqtc.github.io/2012/04/poppler-vs-mupdf.html

Runtimes	Tool			
File	fitz	pdfrw	pdftk	PyPDF2
Adobe.pdf	4,96	20,72	136,34	683,27
Evolution.pdf	0,40	0,41	1,22	0,94
PyMuPDF.pdf	0,04	0,19	1,03	0,97
sdw_2015_01.pdf	0,19	1,19	6,13	6,49
sdw_2015_02.pdf	0,23	1,52	7,74	7,02
sdw_2015_03.pdf	0,39	2,76	13,39	12,67
sdw_2015_04.pdf	0,25	2,14	8,55	7,50
sdw_2015_05.pdf	0,29	1,71	8,92	7,99
sdw_2015_06.pdf	0,53	3,30	16,05	15,56
sdw_2015_07.pdf	0,33	2,17	10,65	10,81
sdw_2015_08.pdf	0,29	2,01	9,65	9,39
sdw_2015_09.pdf	0,36	2,49	11,48	10,97
sdw_2015_10.pdf	0,27	1,87	3,31	6,74
sdw_2015_11.pdf	1,47	12,79	40,18	62,44
sdw_2015_12.pdf	0,39	2,21	10,40	10,19
Total Times	**10,40**	**57,46**	**285,04**	**852,96**

Time Ratios			
1,00	5,52	27,40	81,98
	1,00	4,96	14,84
		1,00	2,99
			1,00

If we leave out the Adobe manual, this table looks like

Runtimes	Tool			
File	fitz	pdfrw	pdftk	PyPDF2
Evolution.pdf	0,40	0,41	1,22	0,94
PyMuPDF.pdf	0,04	0,19	1,03	0,97
sdw_2015_01.pdf	0,19	1,19	6,13	6,49
sdw_2015_02.pdf	0,23	1,52	7,74	7,02
sdw_2015_03.pdf	0,39	2,76	13,39	12,67
sdw_2015_04.pdf	0,25	2,14	8,55	7,50
sdw_2015_05.pdf	0,29	1,71	8,92	7,99
sdw_2015_06.pdf	0,53	3,30	16,05	15,56
sdw_2015_07.pdf	0,33	2,17	10,65	10,81
sdw_2015_08.pdf	0,29	2,01	9,65	9,39
sdw_2015_09.pdf	0,36	2,49	11,48	10,97
sdw_2015_10.pdf	0,27	1,87	3,31	6,74
sdw_2015_11.pdf	1,47	12,79	40,18	62,44
sdw_2015_12.pdf	0,39	2,21	10,40	10,19
Gesamtergebnis	**5,44**	**36,75**	**148,70**	**169,69**

Time Ratios			
1,00	6,75	27,32	31,18
	1,00	4,05	4,62
		1,00	1,14
			1,00

PyMuPDF is by far the fastest: on average 4.5 times faster than the second best (the pure Python tool pdfrw, **chapeau pdfrw!**), and almost 20 times faster than the command line tool pdftk.

Where PyMuPDF only requires less than 13 seconds to process all files, pdftk affords itself almost 4 minutes.

By far the slowest tool is PyPDF2 – it is more than 66 times slower than PyMuPDF and 15 times slower than pdfrw! The main reason for PyPDF2's bad look comes from the Adobe manual. It obviously is slowed down by the linear file structure and the immense amount of bookmarks of this file. If we take out this special case, then PyPDF2 is only 21.5 times slower than PyMuPDF, 4.5 times slower than pdfrw and 1.2 times slower than pdftk.

If we look at the output PDFs, there is one surprise:

Each tool created a PDF of similar size as the original. Apart from the Adobe case, PyMuPDF always created the smallest output.

Adobe's manual is an exception: The pure Python tools pdfrw and PyPDF2 **reduced** its size by more than 20% (and yielded a document which is no longer linearized)!

PyMuPDF and pdftk in contrast **drastically increased** the size by 40% to about 50 MB (also no longer linearized).

So far, we have no explanation of what is happening here.

11.2 Part 2: Text Extraction

We also have compared text extraction speed with other tools.

The following table shows a run time comparison. PyMuPDF's methods appear as "fitz (TEXT)" and "fitz (JSON)" respectively. The tool `pdftotext.exe` of the Xpdf[94] toolset appears as "xpdf".

- **extractText():** basic text extraction without layout re-arrangement (using `GetText(..., output = "text")`)
- **pdftotext:** a command line tool of the **Xpdf** toolset (which also is the basis of Poppler's library[95])
- **extractJSON():** text extraction with layout information (using `GetText(..., output = "json")`)
- **pdfminer:** a pure Python PDF tool specialized on text extraction tasks

All tools have been used with their most basic, fanciless functionality – no layout re-arrangements, etc.

For demonstration purposes, we have included a version of `GetText(doc, output = "json")`, that also re-arranges the output according to occurrence on the page.

Here are the results using the same test files as above (again: decimal point and comma reversed):

Runtime	**Tool**				
File	**1 fitz (TEXT)**	**2 fitz bareJSON**	**3 fitz sortJSON**	**4 xpdf**	**5 pdfminer**
Adobe.pdf	5,16	5,53	6,27	12,42	216,32
Evolution.pdf	0,29	0,29	0,33	1,99	12,91
PyMuPDF.pdf	0,11	0,10	0,12	1,71	4,71
sdw_2015_01.pdf	0,95	0,98	1,12	2,84	43,96
sdw_2015_02.pdf	1,04	1,09	1,14	2,86	48,26
sdw_2015_03.pdf	1,81	1,92	1,97	3,82	153,51
sdw_2015_04.pdf	1,23	1,27	1,37	3,17	80,95
sdw_2015_05.pdf	1,00	1,08	1,15	2,82	48,65
sdw_2015_06.pdf	1,83	1,92	1,98	3,70	138,75
sdw_2015_07.pdf	0,99	1,11	1,16	2,93	55,59
sdw_2015_08.pdf	0,97	1,04	1,12	2,80	48,09
sdw_2015_09.pdf	1,92	1,97	2,05	3,84	159,62
sdw_2015_10.pdf	1,10	1,18	1,25	3,45	74,25
sdw_2015_11.pdf	2,37	2,39	2,50	5,82	166,14
sdw_2015_12.pdf	1,14	1,19	1,26	2,93	69,79
Gesamtergebnis	**21,92**	**23,08**	**24,82**	**57,10**	**1321,51**

1,00	1,05	1,13	2,60	60,28
	1,00	1,08	2,47	57,27
		1,00	2,30	53,24
			1,00	23,15

Again, (Py-) MuPDF is the fastest around. It is 2.3 to 2.6 times faster than xpdf.

`pdfminer`, as a pure Python solution, of course is comparatively slow: MuPDF is 50 to 60 times faster and xpdf is 23 times faster. These observations in order of magnitude coincide with the statements on this web

[94] http://www.foolabs.com/xpdf/
[95] http://poppler.freedesktop.org/

site[96].

11.3 Part 3: Image Rendering

We have tested rendering speed of MuPDF against the `pdftopng.exe`, a command lind tool of the **Xpdf** toolset (the PDF code basis of **Poppler**).

MuPDF invocation using a resolution of 150 pixels (Xpdf default):

```
mutool draw -o t%d.png -r 150 file.pdf
```

PyMuPDF invocation:

```
zoom = 150.0 / 72.0
mat = fitz.Matrix(zoom, zoom)
def ProcessFile(datei):
    print "processing:", datei
    doc=fitz.open(datei)
    for p in fitz.Pages(doc):
        pix = p.getPixmap(matrix=mat, alpha = False)
        pix.writePNG("t-%s.png" % p.number)
        pix = None
    doc.close()
    return
```

Xpdf invocation:

```
pdftopng.exe file.pdf ./
```

The resulting runtimes can be found here (again: meaning of decimal point and comma reversed):

[96] http://www.unixuser.org/~euske/python/pdfminer/

Render Speed	tool		
file	mudraw	pymupdf	xpdf
Adobe.pdf	105,09	110,66	505,27
Evolution.pdf	40,70	42,17	108,33
PyMuPDF.pdf	5,09	4,96	21,82
sdw_2015_01.pdf	29,77	30,40	76,81
sdw_2015_02.pdf	29,67	30,00	74,68
sdw_2015_03.pdf	32,67	32,88	85,89
sdw_2015_04.pdf	30,07	29,59	78,09
sdw_2015_05.pdf	31,37	31,39	77,56
sdw_2015_06.pdf	31,76	31,49	87,89
sdw_2015_07.pdf	33,33	34,58	78,74
sdw_2015_08.pdf	31,83	32,73	75,95
sdw_2015_09.pdf	36,92	36,77	84,37
sdw_2015_10.pdf	30,08	30,48	77,13
sdw_2015_11.pdf	33,21	34,11	80,96
sdw_2015_12.pdf	31,77	32,69	80,68
Gesamtergebnis	**533,33**	**544,90**	**1594,18**

1	1,02	2,99
	1	2,93

- MuPDF and PyMuPDF are both about 3 times faster than Xpdf.
- The 2% speed difference between MuPDF (a utility written in C) and PyMuPDF is the Python overhead.

CHAPTER

TWELVE

APPENDIX 2: DETAILS ON TEXT EXTRACTION

This chapter provides background on the text extraction methods of PyMuPDF.

Information of interest are

- what do they provide?
- what do they imply (processing time / data sizes)?

12.1 General structure of a TextPage

TextPage is one of PyMuPDF's classes. It is normally created behind the curtain, when *Page* text extraction methods are used, but it is also available directly. In any case, an intermediate class, *DisplayList* must be created first (display lists contain interpreted pages, they also provide the input for *Pixmap* creation). Information contained in a *TextPage* has the following hierarchy. Other than its name suggests, images may optionally also be part of a text page:

```
<page>
    <text block>
        <line>
            <span>
                <char>
    <image block>
        <img>
```

A **text page** consists of blocks (= roughly paragraphs).

A **block** consists of either lines and their characters, or an image.

A **line** consists of spans.

A **span** consists of adjacent characters with identical font properties: name, size, flags and color.

12.2 Plain Text

Function *TextPage.extractText()* (or `Page.getText("text")`) extracts a page's plain **text in original order** as specified by the creator of the document (which may not equal a natural reading order).

An example output:

```
>>> print(page.getText("text"))
Some text on first page.
```

12.3 BLOCKS

Function *TextPage.extractBLOCKS()* (or `Page.getText("blocks")`) extracts a page's text blocks as a list of items like:

```
(x0, y0, x1, y1, "lines in block", block_type, block_no)
```

Where the first 4 items are the float coordinates of the block's bbox. The lines within each block are concatenated by a new-line character.

This is a high-speed method with enough information to re-arrange the page's text in natural reading order where required.

Example output:

```
>>> print(page.getText("blocks"))
[(50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375,
'Some text on first page.', 0, 0)]
```

12.4 WORDS

Function *TextPage.extractWORDS()* (or `Page.getText("words")`) extracts a page's text **words** as a list of items like:

```
(x0, y0, x1, y1, "word", block_no, line_no, word_no)
```

Where the first 4 items are the float coordinates of the words's bbox. The last three integers provide some more information on the word's whereabouts.

This is a high-speed method with enough information to extract text contained in a given rectangle.

Example output:

```
>>> for word in page.getText("words"):
        print(word)
(50.0, 88.17500305175781, 78.73200225830078, 103.28900146484375,
'Some', 0, 0, 0)
(81.79000091552734, 88.17500305175781, 99.5219955444336, 103.28900146484375,
'text', 0, 0, 1)
(102.57999420166016, 88.17500305175781, 114.8119888305664, 103.28900146484375,
'on', 0, 0, 2)
(117.86998748779297, 88.17500305175781, 135.5909881591797, 103.28900146484375,
'first', 0, 0, 3)
(138.64898681640625, 88.17500305175781, 166.1709747314453, 103.28900146484375,
'page.', 0, 0, 4)
```

12.5 HTML

TextPage.extractHTML() (or `Page.getText("html")` output fully reflects the structure of the page's `TextPage` – much like DICT / JSON below. This includes images, font information and text positions. If wrapped in HTML header and trailer code, it can readily be displayed by an internate browser. Our above example:

```
>>> for line in page.getText("html").splitlines():
        print(line)

<div id="page0" style="position:relative;width:300pt;height:350pt;
background-color:white">
<p style="position:absolute;white-space:pre;margin:0;padding:0;top:88pt;
left:50pt"><span style="font-family:Helvetica,sans-serif;
font-size:11pt">Some text on first page.</span></p>
</div>
```

12.6 Controlling Quality of HTML Output

While HTML output has improved a lot in MuPDF v1.12.0, it is not yet bug-free: we have found problems in the areas **font support** and **image positioning**.

- HTML text contains references to the fonts used of the original document. If these are not known to the browser (a fat chance!), it will replace them with his assumptions, which probably will let the result look awkward. This issue varies greatly by browser – on my Windows machine, MS Edge worked just fine, whereas Firefox looked horrible.
- For PDFs with a complex structure, images may not be positioned and / or sized correctly. This seems to be the case for rotated pages and pages, where the various possible page bbox variants do not coincide (e.g. `MediaBox != CropBox`). We do not know yet, how to address this – we filed a bug at MuPDF's site.

To address the font issue, you can use a simple utility script to scan through the HTML file and replace font references. Here is a little example that replaces all fonts with one of the *PDF Base 14 Fonts*: serifed fonts will become "Times", non-serifed "Helvetica" and monospaced will become "Courier". Their respective variations for "bold", "italic", etc. are hopefully done correctly by your browser:

```
import sys
filename = sys.argv[1]
otext = open(filename).read()                  # original html text string
pos1 = 0                                       # search start poition
font_serif = "font-family:Times"               # enter ...
font_sans  = "font-family:Helvetica"           # ... your choices ...
font_mono  = "font-family:Courier"             # ... here
found_one  = False                             # true if search successfull

while True:
    pos0 = otext.find("font-family:", pos1)    # start of a font spec
    if pos0 < 0:                               # none found - we are done
        break
    pos1 = otext.find(";", pos0)               # end of font spec
    test = otext[pos0 : pos1]                  # complete font spec string
    testn = ""                                 # the new font spec string
    if test.endswith(",serif"):                # font with serifs?
        testn = font_serif                     # use Times instead
    elif test.endswith(",sans-serif"):         # sans serifs font?
        testn = font_sans                      # use Helvetica
    elif test.endswith(",monospace"):          # monospaced font?
        testn = font_mono                      # becomes Courier

    if testn != "":                            # any of the above found?
        otext = otext.replace(test, testn)     # change the source
```

(continues on next page)

(continued from previous page)

```
        found_one = True
        pos1 = 0                                # start over

if found_one:
    ofile = open(filename + ".html", "w")
    ofile.write(otext)
    ofile.close()
else:
    print("Warning: could not find any font specs!")
```

12.7 DICT (or JSON)

TextPage.extractDICT() (or `Page.getText("dict")`) output fully reflects the structure of a `TextPage` and provides image content and position details (`bbox` – boundary boxes in pixel units) for every block and line. This information can be used to present text in another reading order if required (e.g. from top-left to bottom-right). Images are stored as `bytes` (`bytearray` in Python 2) for DICT output and base64 encoded strings for JSON output.

For a visuallization of the dictionary structure have a look at *Dictionary Structure of extractDICT() and extractRAWDICT()*.

Here is how this looks like:

```
{
    "width": 300.0,
    "height": 350.0,
    "blocks": [{
        "type": 0,
        "bbox": [50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375],
        "lines": [{
            "wmode": 0,
            "dir": [1.0, 0.0],
            "bbox": [50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375],
            "spans": [{
                "size": 11.0,
                "flags": 0,
                "font": "Helvetica",
                "color": 0,
                "text": "Some text on first page.",
                "bbox": [50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375]
            }]
        }]
    }]
}
```

12.8 RAWDICT

TextPage.extractRAWDICT() (or `Page.getText("rawdict")`) is an **information superset of DICT** and takes the detail level one step deeper. It looks exactly like the above, except that the `"text"` items (*string*) are replaced by `"chars"` items (*list*). Each `"chars"` entry is a character *dict*. For example, here is what you would see in place of item `"text": "Text in black color."` above:

```
"chars": [{
    "origin": [50.0, 100.0],
    "bbox": [50.0, 88.17500305175781, 57.336997985839844, 103.28900146484375],
    "c": "S"
}, {
    "origin": [57.33700180053711, 100.0],
    "bbox": [57.33700180053711, 88.17500305175781, 63.4530029296875, 103.28900146484375],
    "c": "o"
}, {
    "origin": [63.4530029296875, 100.0],
    "bbox": [63.4530029296875, 88.17500305175781, 72.61600494384766, 103.28900146484375],
    "c": "m"
}, {
    "origin": [72.61600494384766, 100.0],
    "bbox": [72.61600494384766, 88.17500305175781, 78.73200225830078, 103.28900146484375],
    "c": "e"
}, {
    "origin": [78.73200225830078, 100.0],
    "bbox": [78.73200225830078, 88.17500305175781, 81.79000091552734, 103.28900146484375],
    "c": " "
< ... deleted ... >
}, {
    "origin": [163.11297607421875, 100.0],
    "bbox": [163.11297607421875, 88.17500305175781, 166.1709747314453, 103.28900146484375],
    "c": "."
}],
```

12.9 XML

The *TextPage.extractXML()* (or `Page.getText("xml")`) version extracts text (no images) with the detail level of RAWDICT:

```
>>> for line in page.getText("xml").splitlines():
    print(line)

<page id="page0" width="300" height="350">
<block bbox="50 88.175 166.17098 103.289">
<line bbox="50 88.175 166.17098 103.289" wmode="0" dir="1 0">
<font name="Helvetica" size="11">
<char quad="50 88.175 57.336999 88.175 50 103.289 57.336999 103.289" x="50"
y="100" color="#000000" c="S"/>
<char quad="57.337 88.175 63.453004 88.175 57.337 103.289 63.453004 103.289" x="57.337"
y="100" color="#000000" c="o"/>
<char quad="63.453004 88.175 72.616008 88.175 63.453004 103.289 72.616008 103.289" x="63.453004"
y="100" color="#000000" c="m"/>
<char quad="72.616008 88.175 78.732 88.175 72.616008 103.289 78.732 103.289" x="72.616008"
y="100" color="#000000" c="e"/>
<char quad="78.732 88.175 81.79 88.175 78.732 103.289 81.79 103.289" x="78.732"
y="100" color="#000000" c=" "/>

... deleted ...

<char quad="163.11298 88.175 166.17098 88.175 163.11298 103.289 166.17098 103.289" x="163.11298"
y="100" color="#000000" c="."/>
</font>
```

(continues on next page)

(continued from previous page)

```
</line>
</block>
</page>
```

Note: We have successfully tested lxml[97] to interpret this output.

12.10 XHTML

TextPage.extractXHTML() (or `Page.getText("xhtml")`) is a variation of TEXT but in HTML format, containing the bare text and images ("semantic" output):

```
<div id="page0">
<p>Some text on first page.</p>
</div>
```

12.11 Text Extraction Flags Defaults

New in version 1.16.2: Method *Page.getText()* supports a keyword parameter `flags` *(int)* to control the amount and the quality of extracted data. The following table shows the defaults settings (flags parameter omitted or None) for each extraction variant. A description of the respective bit settings can be found in *Preserve Text Flags*.

Indicator	text	html	xhtml	xml	dict	rawdict	words	blocks
preserve ligatures	1	1	1	1	1	1	1	1
preserve whitespace	1	1	1	1	1	1	1	1
preserve images	n/a	1	1	n/a	1	1	n/a	0
inhibit spaces	0	0	0	0	0	0	0	0

- **"json"** is handled exactly like **"dict"** and is hence left out.
- An "n/a" specification means a value of 0 and setting this bit never has any effect on the output (but an adverse effect on performance).
- If you are not interested in images when using an output variant which includes them by default, then by all means set the respective bit off: You will experience a better performance and much lower space requirements.

To show the effect of `TEXT_INHIBIT_SPACES` have a look at this example:

```
>>> print(page.getText("text"))
H a l l o !
Mo r e  t e x t
i s  f o l l o w i n g
i n  E n g l i s h
. . .  l e t ' s  s e e
w h a t  h a p p e n s .
>>> print(page.getText("text", flags=fitz.TEXT_INHIBIT_SPACES))
```

(continues on next page)

[97] https://pypi.org/project/lxml/

(continued from previous page)

```
Hallo!
More text
is following
in English
... let's see
what happens.
>>>
```

12.12 Performance

The text extraction methods differ significantly: in terms of information they supply, and in terms of resource requirements and runtimes. Generally, more information of course means that more processing is required and a higher data volume is generated.

Note: Especially images have a **very significant** impact. Make sure to exclude them (via the `flags` parameter) whenever you do not need them. To process the below mentioned 2'700 total pages with default flags settings required 160 seconds across all extraction methods. When all images where excluded, less than 50% of that time (77 seconds) were needed.

To begin with, all methods are **very fast** in relation to other products out there in the market. In terms of processing speed, we are not aware of a faster (free) tool. Even the most detailed method, RAWDICT, processes all 1'310 pages of the *Adobe PDF Reference 1.7* in less than 5 seconds (simple text needs less than 2 seconds here).

The following table shows average relative speeds ("RSpeed", baseline 1.00 is TEXT), taken across ca. 1400 text-heavy and 1300 image-heavy pages.

Method	RSpeed	Comments	no images
TEXT	1.00	no images, **plain** text, line breaks	1.00
BLOCKS	1.00	image bboxes (only), **block** level text with bboxes, line breaks	1.00
WORDS	1.02	no images, **word** level text with bboxes	1.02
XML	2.72	no images, **char** level text, layout and font details	2.72
XHTML	3.32	**base64** images, **span** level text, no layout info	1.00
HTML	3.54	**base64** images, **span** level text, layout and font details	1.01
DICT	3.93	**binary** images, **span** level text, layout and font details	1.04
RAWDICT	4.50	**binary** images, **char** level text, layout and font details	1.68

As mentioned: when excluding all images (last column), the relative speeds are changing drastically: except RAWDICT and XML, the other methods are almost equally fast, and RAWDICT requires 40% less execution time than the **now slowest XML**.

Look at chapter **Appendix 1** for more performance information.

CHAPTER

THIRTEEN

APPENDIX 3: CONSIDERATIONS ON EMBEDDED FILES

This chapter provides some background on embedded files support in PyMuPDF.

13.1 General

Starting with version 1.4, PDF supports embedding arbitrary files as part ("Embedded File Streams") of a PDF document file (see chapter 3.10.3, pp. 184 of the *Adobe PDF Reference 1.7*).

In many aspects, this is comparable to concepts also found in ZIP files or the OLE technique in MS Windows. PDF embedded files do, however, *not* support directory structures as does the ZIP format. An embedded file can in turn contain embedded files itself.

Advantages of this concept are that embedded files are under the PDF umbrella, benefitting from its permissions / password protection and integrity aspects: all data, which a PDF may reference or even may be dependent on, can be bundled into it and so form a single, consistent unit of information.

In addition to embedded files, PDF 1.7 adds *collections* to its support range. This is an advanced way of storing and presenting meta information (i.e. arbitrary and extensible properties) of embedded files.

13.2 MuPDF Support

After adding initial support for collections (portfolios) and `/EmbeddedFiles` in MuPDF version 1.11, this support was dropped again in version 1.15.

As a consequence, the cli utility `mutool` no longer offers access to embedded files.

PyMuPDF – having implemented an `/EmbeddedFiles` API in response in its version 1.11.0 – was therefore forced to change gears starting with its version 1.16.0 (we never published a MuPDF v1.15.x compatible PyMuPDF).

We are now maintaining our own code basis supporting embedded files. This code makes use of basic MuPDF dictionary and array functions only.

13.3 PyMuPDF Support

We continue to support the full old API with respect to embedded files – with only minor, cosmetic changes.

There even also is a new function, which delivers a list of all names under which embedded data are resgistered in a PDF, *Document.embeddedFileNames()*.

CHAPTER

FOURTEEN

APPENDIX 4: ASSORTED TECHNICAL INFORMATION

14.1 PDF Base 14 Fonts

The following 14 builtin font names **must be supported by every PDF viewer** application. They are available as a dictionary, which maps their full names amd their abbreviations in lower case to the full font basename. Whereever a **fontname** must be provided in PyMuPDF, any **key or value** from the dictionary may be used:

```
In [2]: fitz.Base14_fontdict
Out[2]:
{'courier': 'Courier',
'courier-oblique': 'Courier-Oblique',
'courier-bold': 'Courier-Bold',
'courier-boldoblique': 'Courier-BoldOblique',
'helvetica': 'Helvetica',
'helvetica-oblique': 'Helvetica-Oblique',
'helvetica-bold': 'Helvetica-Bold',
'helvetica-boldoblique': 'Helvetica-BoldOblique',
'times-roman': 'Times-Roman',
'times-italic': 'Times-Italic',
'times-bold': 'Times-Bold',
'times-bolditalic': 'Times-BoldItalic',
'symbol': 'Symbol',
'zapfdingbats': 'ZapfDingbats',
'helv': 'Helvetica',
'heit': 'Helvetica-Oblique',
'hebo': 'Helvetica-Bold',
'hebi': 'Helvetica-BoldOblique',
'cour': 'Courier',
'coit': 'Courier-Oblique',
'cobo': 'Courier-Bold',
'cobi': 'Courier-BoldOblique',
'tiro': 'Times-Roman',
'tibo': 'Times-Bold',
'tiit': 'Times-Italic',
'tibi': 'Times-BoldItalic',
'symb': 'Symbol',
'zadb': 'ZapfDingbats'}
```

In contrast to their obligation, not all PDF viewers support these fonts correctly and completely – this is especially true for Symbol and ZapfDingbats. Also, the glyph (visual) images will be specific to every reader.

To see how these fonts can be used – including the **CJK built-in** fonts – look at the table in *Page.insertFont()*.

14.2 Adobe PDF Reference 1.7

This PDF Reference manual published by Adobe is frequently quoted throughout this documentation. It can be viewed and downloaded from here[98].

14.3 Using Python Sequences as Arguments in PyMuPDF

When PyMuPDF objects and methods require a Python **list** of numerical values, other Python **sequence types** are also allowed. Python classes are said to implement the **sequence protocol**, if they have a `__getitem__()` method.

This basically means, you can interchangeably use Python `list` or `tuple` or even `array.array`, `numpy.array` and `bytearray` types in these cases.

For example, specifying a sequence `"s"` in any of the following ways

- `s = [1, 2]`
- `s = (1, 2)`
- `s = array.array("i", (1, 2))`
- `s = numpy.array((1, 2))`
- `s = bytearray((1, 2))`

will make it usable in the following example expressions:

- `fitz.Point(s)`
- `fitz.Point(x, y) + s`
- `doc.select(s)`

Similarly with all geometry objects *Rect*, *IRect*, *Matrix* and *Point*.

Because all PyMuPDF geometry classes themselves are special cases of sequences, they (with the exception of *Quad* – see below) can be freely used where numerical sequences can be used, e.g. as arguments for functions like `list()`, `tuple()`, `array.array()` or `numpy.array()`. Look at the following snippet to see this work.

```
>>> import fitz, array, numpy as np
>>> m = fitz.Matrix(1, 2, 3, 4, 5, 6)
>>>
>>> list(m)
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
>>>
>>> tuple(m)
(1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
>>>
>>> array.array("f", m)
array('f', [1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
```

(continues on next page)

[98] http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf

(continued from previous page)

```
>>>
>>> np.array(m)
array([1., 2., 3., 4., 5., 6.])
```

Note: *Quad* is a Python sequence object as well and has a length of 4. Its items however are *point_like* – not numbers. Therefore, the above remarks do not apply.

14.4 Ensuring Consistency of Important Objects in PyMuPDF

PyMuPDF is a Python binding for the C library MuPDF. While a lot of effort has been invested by MuPDF's creators to approximate some sort of an object-oriented behavior, they certainly could not overcome basic shortcomings of the C language in that respect.

Python on the other hand implements the OO-model in a very clean way. The interface code between PyMuPDF and MuPDF consists of two basic files: `fitz.py` and `fitz_wrap.c`. They are created by the excellent SWIG tool for each new version.

When you use one of PyMuPDF's objects or methods, this will result in excution of some code in `fitz.py`, which in turn will call some C code compiled with `fitz_wrap.c`.

Because SWIG goes a long way to keep the Python and the C level in sync, everything works fine, if a certain set of rules is being strictly followed. For example: **never access** a *Page* object, after you have closed (or deleted or set to `None`) the owning *Document*. Or, less obvious: **never access** a page or any of its children (links or annotations) after you have executed one of the document methods `select()`, `deletePage()`, `insertPage()` ... and more.

But just no longer accessing invalidated objects is actually not enough: They should rather be actively deleted entirely, to also free C-level resources (meaning allocated memory).

The reason for these rules lies in the fact that there is a hierachical 2-level one-to-many relationship between a document and its pages and also between a page and its links / annotations. To maintain a consistent situation, any of the above actions must lead to a complete reset – in **Python and, synchronously, in C**.

SWIG cannot know about this and consequently does not do it.

The required logic has therefore been built into PyMuPDF itself in the following way.

1. If a page "loses" its owning document or is being deleted itself, all of its currently existing annotations and links will be made unusable in Python, and their C-level counterparts will be deleted and deallocated.
2. If a document is closed (or deleted or set to `None`) or if its structure has changed, then similarly all currently existing pages and their children will be made unusable, and corresponding C-level deletions will take place. "Structure changes" include methods like `select()`, `delePage()`, `insertPage()`, `insertPDF()` and so on: all of these will result in a cascade of object deletions.

The programmer will normally not realize any of this. If he, however, tries to access invalidated objects, exceptions will be raised.

Invalidated objects cannot be directly deleted as with Python statements like `del page` or `page = None`, etc. Instead, their `__del__` method must be invoked.

All pages, links and annotations have the property `parent`, which points to the owning object. This is the property that can be checked on the application level: if `obj.parent == None` then the object's parent is gone, and any reference to its properties or methods will raise an exception informing about this "orphaned" state.

A sample session:

```
>>> page = doc[n]
>>> annot = page.firstAnnot
>>> annot.type                    # everything works fine
[5, 'Circle']
>>> page = None                   # this turns 'annot' into an orphan
>>> annot.type
<... omitted lines ...>
RuntimeError: orphaned object: parent is None
>>>
>>> # same happens, if you do this:
>>> annot = doc[n].firstAnnot     # deletes the page again immediately!
>>> annot.type                    # so, 'annot' is 'born' orphaned
<... omitted lines ...>
RuntimeError: orphaned object: parent is None
```

This shows the cascading effect:

```
>>> doc = fitz.open("some.pdf")
>>> page = doc[n]
>>> annot = page.firstAnnot
>>> page.rect
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>> annot.type
[5, 'Circle']
>>> del doc                       # or doc = None or doc.close()
>>> page.rect
<... omitted lines ...>
RuntimeError: orphaned object: parent is None
>>> annot.type
<... omitted lines ...>
RuntimeError: orphaned object: parent is None
```

Note: Objects outside the above relationship are not included in this mechanism. If you e.g. created a table of contents by `toc = doc.getToC()`, and later close or change the document, then this cannot and does not change variable `toc` in any way. It is your responsibility to refresh such variables as required.

14.5 Design of Method `Page.showPDFpage()`

14.5.1 Purpose and Capabilities

The method displays an image of a ("source") page of another PDF document within a specified rectangle of the current ("containing", "target") page.

- **In contrast** to *`Page.insertImage()`*, this display is vector-based and hence remains accurate across zooming levels.

- **Just like** *Page.insertImage()*, the size of the display is adjusted to the given rectangle.

The following variations of the display are currently supported:

- Bool parameter `keep_proportion` controls whether to maintain the aspect ratio (default) or not.
- Rectangle parameter `clip` restricts the visible part of the source page rectangle. Default is the full page.
- float `rotation` rotates the display by an arbitrary angle (degrees). If the angle is not an integer multiple of 90, only 2 of the 4 corners may be positioned on the target border if also `keep_proportion` is true.
- Bool parameter `overlay` controls whether to put the image on top (foreground, default) of current page content or not (background).

Use cases include (but are not limited to) the following:

1. "Stamp" a series of pages of the current document with the same image, like a company logo or a watermark.
2. Combine arbitrary input pages into one output page to support "booklet" or double-sided printing (known as "4-up", "n-up").
3. Split up (large) input pages into several arbitrary pieces. This is also called "posterization", because you e.g. can split an A4 page horizontally and vertically, print the 4 pieces enlarged to separate A4 pages, and end up with an A2 version of your original page.

14.5.2 Technical Implementation

This is done using PDF **"Form XObjects"**, see section 4.9 on page 355 of *Adobe PDF Reference 1.7*. On execution of a `Page.showPDFpage(rect, src, pno, ...)`, the following things happen:

1. The *resources* and *contents* objects of page `pno` in document `src` are copied over to the current document, jointly creating a new **Form XObject** with the following properties. The PDF *xref* number of this object is returned by the method.

    a. `/BBox` equals `/Mediabox` of the source page

    b. `/Matrix` equals the identity matrix `[1 0 0 1 0 0]`

    c. `/Resources` equals that of the source page. This involves a "deep-copy" of hierarchically nested other objects (including fonts, images, etc.). The complexity involved here is covered by MuPDF's grafting[99] technique functions.

    d. This is a stream object type, and its stream is an exact copy of the combined data of the source page's `/Contents` objects.

    This step is only executed once per shown source page. Subsequent displays of the same page only create pointers (done in next step) to this object.

[99] MuPDF supports "deep-copying" objects between PDF documents. To avoid duplicate data in the target, it uses so-called "graftmaps", like a form of scratchpad: for each object to be copied, its *xref* number is looked up in the graftmap. If found, copying is skipped. Otherwise, the new *xref* is recorded and the copy takes place. PyMuPDF makes use of this technique in two places so far: *Document.insertPDF()* and *Page.showPDFpage()*. This process is fast and very efficient, because it prevents multiple copies of typically large and frequently referenced data, like images and fonts. However, you may still want to consider using garbage collection (option 4) in any of the following cases:

1. The target PDF is not new / empty: grafting does not check for resource types that already existed (e.g. images, fonts) in the target document
2. Using *Page.showPDFpage()* for more than one source document: each grafting occurs **within one source** PDF only, not across multiple.

2. A second **Form XObject** is then created which the target page uses to invoke the display. This object has the following properties:
    a. `/BBox` equals the `/CropBox` of the source page (or `clip`).
    b. `/Matrix` represents the mapping of `/BBox` to the target rectangle.
    c. `/XObject` references the previous XObject via the fixed name `fullpage`.
    d. The stream of this object contains exactly one fixed statement: `/fullpage Do`.
3. The *`resources`* and *`contents`* objects of the target page are now modified as follows.
    a. Add an entry to the `/XObject` dictionary of `/Resources` with the name `fzFrm<n>` (with n chosen such that this entry is unique on the page).
    b. Depending on `overlay`, prepend or append a new object to the page's `/Contents` array, containing the statement `q /fzFrm<n> Do Q`.

14.6 Redirecting Error and Warning Messages

Since MuPDF version 1.16 error and warning messages can be redirected via an official plugin.

PyMuPDF will put error messages to `sys.stderr` prefixed with the string "mupdf:". Warnings are internally stored and can be accessed via `fitz.TOOLS.mupdf_warnings()`. There also is a function to empty this store.

CHAPTER

FIFTEEN

CHANGE LOGS

15.1 Changes in Version 1.16.7

Minor changes to better synchronize the binary image streams of *TextPage* image blocks and *Document.extractImage()* images.

- **Fixed** issue #394 ("PyMuPDF Segfaults when using TOOLS.mupdf_warnings()").
- **Changed** redirection of MuPDF error messages: apart from writing them to Python `sys.stderr`, they are now also stored with the MuPDF warnings.
- **Changed** *Tools.mupdf_warnings()* to automatically empty the store (if not deactivated via a parameter).
- **Changed** *Page.getImageBbox()* to return an **infinite rectangle** if the image could not be located on the page – instead of raising an exception.

15.2 Changes in Version 1.16.6

- **Fixed** issue #390 ("Incomplete deletion of annotations").
- **Changed** *Page.searchFor()* / *Document.searchPageFor()* to also support the `flags` parameter, which controls the data included in a *TextPage*.
- **Changed** *Document.getPageImageList()*, *Document.getPageFontList()* and their *Page* counterparts to support a new parameter `full`. If true, the returned items will contain the *xref* of the *Form XObject* where the font or image is referenced.

15.3 Changes in Version 1.16.5

More performance improvements for text extraction.

- **Fixed** second part of issue #381 (see item in v1.16.4).
- **Added** *Page.getTextPage()*, so it is no longer required to create an intermediate display list for text extractions. Page level wrappers for text extraction and text searching are now based on this, which should improve performance by ca. 5%.

15.4 Changes in Version 1.16.4

- **Fixed** issue #381 ("TextPage.extractDICT ... failed ... after upgrading ... to 1.16.3")
- **Added** method *Document.pages()* which delivers a generator iterator over a page range.
- **Added** method *Page.links()* which delivers a generator iterator over the links of a page.
- **Added** method *Page.annots()* which delivers a generator iterator over the annotations of a page.
- **Added** method *Page.widgets()* which delivers a generator iterator over the form fields of a page.
- **Changed** *Document.isFormPDF* to now contain the number of widgets, and `False` if not a PDF or this number is zero.

15.5 Changes in Version 1.16.3

Minor changes compared to version 1.16.2. The code of the "dict" and "rawdict" variants of *Page.getText()* has been ported to C which has greatly improved their performance. This improvement is mostly noticeable with text-oriented documents, where they now should execute almost two times faster.

- **Fixed** issue #369 ("mupdf: cmsCreateTransform failed") by removing ICC colorspace support.
- **Changed** *Page.getText()* to accept additional keywords "blocks" and "words". These will deliver the results of *Page.getTextBlocks()* and *Page.getTextWords()*, respectively. So all text extraction methods are now available via a uniform API. Correspondingly, there are now new methods *TextPage.extractBLOCKS()* and `TextPage.extractWords()`.
- **Changed** *Page.getText()* to default bit indicator `TEXT_INHIBIT_SPACES` to **off**. Insertion of additional spaces is **not suppressed** by default.

15.6 Changes in Version 1.16.2

- **Changed** text extraction methods of *Page* to allow detail control of the amount of extracted data.
- **Added** *planishLine()* which maps a given line (defined as a pair of points) to the x-axis.
- **Fixed** an issue (w/o Github number) which brought down the interpreter when encountering certain non-UTF-8 encodable characters while using *Page.getText()* with te "dict" option.
- **Fixed** issue #362 ("Memory Leak with getText('rawDICT')").

15.7 Changes in Version 1.16.1

- **Added** property *Quad.isConvex* which checks whether a line is contained in the quad if it connects two points of it.
- **Changed** *Document.insertPDF()* to now allow dropping or including links and annotations independently during the copy. Fixes issue #352 ("Corrupt PDF data and ..."), which seemed to intermittently occur when using the method for some problematic PDF files.
- **Fixed** a bug which, in matrix division using the syntax `"m1/m2"`, caused matrix `"m1"` to be **replaced** by the result instead of delivering a new matrix.

- **Fixed** issue #354 ("SyntaxWarning with Python 3.8"). We now always use `"=="` for literals (instead of the `"is"` Python keyword).
- **Fixed** issue #353 ("mupdf version check"), to no longer refuse the import when there are only patch level deviations from MuPDF.

15.8 Changes in Version 1.16.0

This major new version of MuPDF comes with several nice new or changed features. Some of them imply programming API changes, however. This is a synopsis of what has changed:

- PDF document encryption and decryption is now **fully supported**. This includes setting **permissions**, **passwords** (user and owner passwords) and the desired encryption method.
- In response to the new encryption features, PyMuPDF returns an integer (ie. a combination of bits) for document permissions, and no longer a dictionary.
- Redirection of MuPDF errors and warnings is now natively supported. PyMuPDF redirects error messages from MuPDF to `sys.stderr` and no longer buffers them. Warnings continue to be buffered and will not be displayed. Functions exist to access and reset the warnings buffer.
- Annotations are now **only supported for PDF**.
- Annotations and widgets (form fields) are now **separate object chains** on a page (although widgets technically still **are** PDF annotations). This means, that you will **never encounter widgets** when using *Page.firstAnnot* or *Annot.next()*. You must use *Page.firstWidget* and *Widget.next()* to access form fields.
- As part of MuPDF's changes regarding widgets, only the following four fonts are supported, when **adding** or **changing** form fields: **Courier, Helvetica, Times-Roman** and **ZapfDingBats**.

List of change details:

- **Added** *Document.can_save_incrementally()* which checks conditions that are preventing use of option `incremental=True` of *Document.save()*.
- **Added** *Page.firstWidget* which points to the first field on a page.
- **Added** *Page.getImageBbox()* which returns the rectangle occupied by an image shown on the page.
- **Added** *Annot.setName()* which lets you change the (icon) name field.
- **Added** outputting the text color in *Page.getText()*: the `"dict"`, `"rawdict"` and `"xml"` options now also show the color in sRGB format.
- **Changed** *Document.permissions* to now contain an integer of bool indicators – was a dictionary before.
- **Changed** *Document.save()*, *Document.write()*, which now fully support password-based decryption and encryption of PDF files.
- **Changed the names of all Python constants** related to annotations and widgets. Please make sure to consult the **Constants and Enumerations** chapter if your script is dealing with these two classes. This decision goes back to the dropped support for non-PDF annotations. The **old names** (starting with "ANNOT_*" or "WIDGET_*") will be available as deprecated synonyms.
- **Changed** font support for widgets: only `Cour` (Courier), `Helv` (Helvetica, default), `TiRo` (Times-Roman) and `ZaDb` (ZapfDingBats) are accepted when **adding or changing** form fields. Only the plain versions are possible – not their italic or bold variations. **Reading** widgets, however will show its original font.

- **Changed** the name of the warnings buffer to *Tools.mupdf_warnings()* and the function to empty this buffer is now called *Tools.reset_mupdf_warnings()*.
- **Changed** *Page.getPixmap()*, *Document.getPagePixmap()*: a new bool argument `annots` can now be used to **suppress the rendering of annotations** on the page.
- **Changed** *Page.addFileAnnot()* and *Page.addTextAnnot()* to enable setting an icon.
- **Removed** widget-related methods and attributes from the *Annot* object.
- **Removed** *Document* attributes `openErrCode`, `openErrMsg`, and *Tools* attributes / methods `stderr`, `reset_stderr`, `stdout`, and `reset_stdout`.
- **Removed thirdparty zlib** dependency in PyMuPDF: there are now compression functions available in MuPDF. Source installers of PyMuPDF may now omit this extra installation step.

15.9 No version published for MuPDF v1.15.0

15.10 Changes in Version 1.14.20 / 1.14.21

- **Changed** text marker annotations to support multiple rectangles / quadrilaterals. This fixes issue #341 ("Question : How to addhighlight so that a string spread across more than a line is covered by one highlight?") and similar (#285).
- **Fixed** issue #331 ("Importing PyMuPDF changes warning filtering behaviour globally").

15.11 Changes in Version 1.14.19

- **Fixed** issue #319 ("InsertText function error when use custom font").
- **Added** new method *Document.getSigFlags()* which returns information on whether a PDF is signed. Resolves issue #326 ("How to detect signature in a form pdf?").

15.12 Changes in Version 1.14.17

- **Added** *Document.fullcopyPage()* to make full page copies within a PDF (not just copied references as *Document.copyPage()* does).
- **Changed** *Page.getPixmap()*, *Document.getPagePixmap()* now use `alpha=False` as default.
- **Changed** text extraction: the span dictionary now (again) contains its rectangle under the `bbox` key.
- **Changed** *Document.movePage()* and *Document.copyPage()* to use direct functions instead of wrapping *Document.select()* – similar to *Document.deletePage()* in v1.14.16.

15.13 Changes in Version 1.14.16

- **Changed** *Document* methods around PDF `/EmbeddedFiles` to no longer use MuPDF's "portfolio" functions. That support will be dropped in MuPDF v1.15 – therefore another solution was required.
- **Changed** *Document.embeddedFileCount()* to be a function (was an attribute).

- **Added** new method *Document.embeddedFileNames()* which returns a list of names of embedded files.
- **Changed** *Document.deletePage()* and *Document.deletePageRange()* to internally no longer use *Document.select()*, but instead use functions to perform the deletion directly. As it has turned out, the *Document.select()* method yields invalid outline trees (tables of content) for very complex PDFs and sophisticated use of annotations.

15.14 Changes in Version 1.14.15

- **Fixed** issues #301 ("Line cap and Line join"), #300 ("How to draw a shape without outlines") and #298 ("utils.updateRect exception"). These bugs pertain to drawing shapes with PyMuPDF. Drawing shapes without any border is fully supported. Line cap styles and line line join style are now differentiated and support all possible PDF values (0, 1, 2) instead of just being a bool. The previous parameter `roundCap` is deprecated in favor of `lineCap` and `lineJoin` and will be deleted in the next release.
- **Fixed** issue #290 ("Memory Leak with getText('rawDICT')"). This bug caused memory not being (completely) freed after invoking the "dict", "rawdict" and "json" versions of *Page.getText()*.

15.15 Changes in Version 1.14.14

- **Added** new low-level function *ImageProperties()* to determine a number of characteristics for an image.
- **Added** new low-level function *Document.isStream()*, which checks whether an object is of stream type.
- **Changed** low-level functions *Document._getXrefString()* and *Document._getTrailerString()* now by default return object definitions in a formatted form which makes parsing easy.

15.16 Changes in Version 1.14.13

- **Changed** methods working with binary input: while ever supporting bytes and bytearray objects, they now also accept `io.BytesIO` input, using their `getvalue()` method. This pertains to document creation, embedded files, FileAttachment annotations, pixmap creation and others. Fixes issue #274 ("Segfault when using BytesIO as a stream for insertImage").
- **Fixed** issue #278 ("Is insertImage(keep_proportion=True) broken?"). Images are now correctly presented when keeping aspect ratio.

15.17 Changes in Version 1.14.12

- **Changed** the draw methods of *Page* and *Shape* to support not only RGB, but also GRAY and CMYK colorspaces. This solves issue #270 ("Is there a way to use CMYK color to draw shapes?"). This change also applies to text insertion methods of *Shape*, resp. *Page*.
- **Fixed** issue #269 ("AttributeError in Document.insertPage()"), which occurred when using *Document.insertPage()* with text insertion.

15.18 Changes in Version 1.14.11

- **Changed** *Page.showPDFpage()* to always position the source rectangle centered in the target. This method now also supports **rotation by arbitrary angles**. The argument `reuse_xref` has been deprecated: prevention of duplicates is now **handled internally**.
- **Changed** *Page.insertImage()* to support rotated display of the image and keeping the aspect ratio. Only rotations by multiples of 90 degrees are supported here.
- **Fixed** issue #265 ("TypeError: insertText() got an unexpected keyword argument 'idx'"). This issue only occurred when using *Document.insertPage()* with also inserting text.

15.19 Changes in Version 1.14.10

- **Changed** *Page.showPDFpage()* to support rotation of the source rectangle. Fixes #261 ("Cannot rotate insterted pages").
- **Fixed** a bug in *Page.insertImage()* which prevented insertion of multiple images provided as streams.

15.20 Changes in Version 1.14.9

- **Added** new low-level method *Document._getTrailerString()*, which returns the trailer object of a PDF. This is much like *Document._getXrefString()* except that the PDF trailer has no / needs no *xref* to identify it.
- **Added** new parameters for text insertion methods. You can now set stroke and fill colors of glyphs (text characters) independently, as well as the thickness of the glyph border. A new parameter `render_mode` controls the use of these colors, and whether the text should be visible at all.
- **Fixed** issue #258 ("Copying image streams to new PDF without size increase"): For JPX images embedded in a PDF, *Document.extractImage()* will now return them in their original format. Previously, the MuPDF base library was used, which returns them in PNG format (entailing a massive size increase).
- **Fixed** issue #259 ("Morphing text to fit inside rect"). Clarified use of *getTextlength()* and removed extra line breaks for long words.

15.21 Changes in Version 1.14.8

- **Added** *Pixmap.setRect()* to change the pixel values in a rectangle. This is also an alternative to setting the color of a complete pixmap (*Pixmap.clearWith()*).
- **Fixed** an image extraction issue with JBIG2 (monochrome) encoded PDF images. The issue occurred in *Page.getText()* (parameters "dict" and "rawdict") and in *Document.extractImage()* methods.
- **Fixed** an issue with not correctly clearing a non-alpha *Pixmap* (*Pixmap.clearWith()*).
- **Fixed** an issue with not correctly inverting colors of a non-alpha *Pixmap* (*Pixmap.invertIRect()*).

15.22 Changes in Version 1.14.7

- **Added** *Pixmap.setPixel()* to change one pixel value.
- **Added** documentation for image conversion in the *Collection of Recipes*.
- **Added** new function *getTextlength()* to determine the string length for a given font.
- **Added** Postscript image output (changed *Pixmap.writeImage()* and *Pixmap.getImageData()*).
- **Changed** *Pixmap.writeImage()* and *Pixmap.getImageData()* to ensure valid combinations of colorspace, alpha and output format.
- **Changed** *Pixmap.writeImage()*: the desired format is now inferred from the filename.
- **Changed** FreeText annotations can now have a transparent background - see *Annot.update()*.

15.23 Changes in Version 1.14.5

- **Changed:** *Shape* methods now strictly use the transformation matrix of the *Page* – instead of "manually" calculating locations.
- **Added** method *Pixmap.pixel()* which returns the pixel value (a list) for given pixel coordinates.
- **Added** method *Pixmap.getImageData()* which returns a bytes object representing the pixmap in a variety of formats. Previously, this could be done for PNG outputs only (*Pixmap.getPNGData()*).
- **Changed:** output of methods *Pixmap.writeImage()* and (the new) *Pixmap.getImageData()* may now also be PSD (Adobe Photoshop Document).
- **Added** method *Shape.drawQuad()* which draws a *Quad*. This actually is a shorthand for a *Shape.drawPolyline()* with the edges of the quad.
- **Changed** method *Shape.drawOval()*: the argument can now be **either** a rectangle (*rect_like*) **or** a quadrilateral (*quad_like*).

15.24 Changes in Version 1.14.4

- **Fixes** issue #239 "Annotation coordinate consistency".

15.25 Changes in Version 1.14.3

This patch version contains minor bug fixes and CJK font output support.

- **Added** support for the four CJK fonts as PyMuPDF generated text output. This pertains to methods *Page.insertFont()*, *Shape.insertText()*, *Shape.insertTextbox()*, and corresponding *Page* methods. The new fonts are available under "reserved" fontnames "china-t" (traditional Chinese), "china-s" (simplified Chinese), "japan" (Japanese), and "korea" (Korean).
- **Added** full support for the built-in fonts 'Symbol' and 'Zapfdingbats'.
- **Changed:** The 14 standard fonts can now each be referenced by a 4-letter abbreviation.

15.26 Changes in Version 1.14.1

This patch version contains minor performance improvements.

- **Added** support for *Document* filenames given as `pathlib` object by using the Python `str()` function.

15.27 Changes in Version 1.14.0

To support MuPDF v1.14.0, massive changes were required in PyMuPDF – most of them purely technical, with little visibility to developers. But there are also quite a lot of interesting new and improved features. Following are the details:

- **Added** "ink" annotation.
- **Added** "rubber stamp" annotation.
- **Added** "squiggly" text marker annotation.
- **Added** new class *Quad* (quadrilateral or tetragon) – which represents a general four-sided shape in the plane. The special subtype of rectangular, non-empty tetragons is used in text marker annotations and as returned objects in text search methods.
- **Added** a new option "decrypt" to *Document.save()* and *Document.write()*. Now you can **keep encryption** when saving a password protected PDF.
- **Added** suppression and redirection of unsolicited messages issued by the underlying C-library MuPDF. Consult *Redirecting Error and Warning Messages* for details.
- **Changed:** Changes to annotations now **always require** *Annot.update()* to become effective.
- **Changed** free text annotations to support the full Latin character set and range of appearance options.
- **Changed** text searching, *Page.searchFor()*, to optionally return *Quad* instead *Rect* objects surrounding each search hit.
- **Changed** plain text output: we now add a `\n` to each line if it does not itself end with this character.
- **Fixed** issue 211 ("Something wrong in the doc").
- **Fixed** issue 213 ("Rewritten outline is displayed only by mupdf-based applications").
- **Fixed** issue 214 ("PDF decryption GONE!").
- **Fixed** issue 215 ("Formatting of links added with pyMuPDF").
- **Fixed** issue 217 ("extraction through json is failing for my pdf").

Behind the curtain, we have changed the implementation of geometry objects: they now purely exist in Python and no longer have "shadow" twins on the C-level (in MuPDF). This has improved processing speed in that area by more than a factor of two.

Because of the same reason, most methods involving geometry parameters now also accept the corresponding Python sequence. For example, in method `"page.showPDFpage(rect, ...)"` parameter `rect` may now be any *rect_like* sequence.

We also invested considerable effort to further extend and improve the *Collection of Recipes* chapter.

15.28 Changes in Version 1.13.19

This version contains some technical / performance improvements and bug fixes.

- **Changed** memory management: for Python 3 builds, Python memory management is exclusively used across all C-level code (i.e. no more native `malloc()` in MuPDF code or PyMuPDF interface code). This leads to improved memory usage profiles and also some runtime improvements: we have seen > 2% shorter runtimes for text extractions and pixmap creations (on Windows machines only to date).
- **Fixed** an error occurring in Python 2.7, which crashed the interpreter when using *TextPage.extractRAWDICT()* (= `Page.getText("rawdict")`).
- **Fixed** an error occurring in Python 2.7, when creating link destinations.
- **Extended** the *Collection of Recipes* chapter with more examples.

15.29 Changes in Version 1.13.18

- **Added** method *TextPage.extractRAWDICT()*, and a corresponding new string parameter "rawdict" to method *Page.getText()*. It extracts text and images from a page in Python *dict* form like *TextPage.extractDICT()*, but with the detail level of *TextPage.extractXML()*, which is position information down to each single character.

15.30 Changes in Version 1.13.17

- **Fixed** an error that intermittently caused an exception in *Page.showPDFpage()*, when pages from many different source PDFs were shown.
- **Changed** method *Document.extractImage()* to now return more meta information about the extracted imgage. Also, its performance has been greatly improved. Several demo scripts have been changed to make use of this method.
- **Changed** method *Document._getXrefStream()* to now return `None` if the object is no stream and no longer raise an exception if otherwise.
- **Added** method *Document._deleteObject()* which deletes a PDF object identified by its *xref*. Only to be used by the experienced PDF expert.
- **Added** a method *PaperRect()* which returns a *Rect* for a supplied paper format string. Example: `fitz.PaperRect("letter") = fitz.Rect(0.0, 0.0, 612.0, 792.0)`.
- **Added** a *Collection of Recipes* chapter to this document.

15.31 Changes in Version 1.13.16

- **Added** support for correctly setting transparency (opacity) for certain annotation types.
- **Added** a tool property (*Tools.fitz_config*) showing the configuration of this PyMuPDF version.
- **Fixed** issue #193 ('insertText(overlay=False) gives "cannot resize a buffer with shared storage" error') by avoiding read-only buffers.

15.32 Changes in Version 1.13.15

- **Fixed** issue #189 ("cannot find builtin CJK font"), so we are supporting builtin CJK fonts now (CJK = China, Japan, Korea). This should lead to correctly generated pixmaps for documents using these languages. This change has consequences for our binary file size: it will now range between 8 and 10 MB, depending on the OS.
- **Fixed** issue #191 ("Jupyter notebook kernel dies after ca. 40 pages"), which occurred when modifying the contents of an annotation.

15.33 Changes in Version 1.13.14

This patch version contains several improvements, mainly for annotations.

- **Changed** *Annot.lineEnds* is now a list of two integers representing the line end symbols. Previously was a *dict* of strings.
- **Added** support of line end symbols for applicable annotations. PyMuPDF now can generate these annotations including the line end symbols.
- **Added** *Annot.setLineEnds()* adds line end symbols to applicable annotation types ('Line', 'PolyLine', 'Polygon').
- **Changed** technical implementation of *Page.insertImage()* and *Page.showPDFpage()*: they now create there own contents objects, thereby avoiding changes of potentially large streams with consequential compression / decompression efforts and high change volumes with incremental updates.

15.34 Changes in Version 1.13.13

This patch version contains several improvements for embedded files and file attachment annotations.

- **Added** *Document.embeddedFileUpd()* which allows changing **file content and metadata** of an embedded file. It supersedes the old method *Document.embeddedFileSetInfo()* (which will be deleted in a future version). Content is automatically compressed and metadata may be unicode.
- **Changed** *Document.embeddedFileAdd()* to now automatically compress file content. Accompanying metadata can now be unicode (had to be ASCII in the past).
- **Changed** *Document.embeddedFileDel()* to now automatically delete **all entries** having the supplied identifying name. The return code is now an integer count of the removed entries (was `None` previously).
- **Changed** embedded file methods to now also accept or show the PDF unicode filename as additional parameter `ufilename`.
- **Added** *Page.addFileAnnot()* which adds a new file attachment annotation.
- **Changed** *Annot.fileUpd()* (file attachment annot) to now also accept the PDF unicode `ufilename` parameter. The description parameter `desc` correctly works with unicode. Furthermore, **all** parameters are optional, so metadata may be changed without also replacing the file content.
- **Changed** *Annot.fileInfo()* (file attachment annot) to now also show the PDF unicode filename as parameter `ufilename`.
- **Fixed** issue #180 ("page.getText(output='dict') return invalid bbox") to now also work for vertical text.

- **Fixed** issue #185 ("Can't render the annotations created by PyMuPDF"). The issue's cause was the minimalistic MuPDF approach when creating annotations. Several annotation types have no `/AP` ("appearance") object when created by MuPDF functions. MuPDF, SumatraPDF and hence also PyMuPDF cannot render annotations without such an object. This fix now ensures, that an appearance object is always created together with the annotation itself. We still do not support line end styles.

15.35 Changes in Version 1.13.12

- **Fixed** issue #180 ("page.getText(output='dict') return invalid bbox"). Note that this is a circumvention of an MuPDF error, which generates zero-height character rectangles in some cases. When this happens, this fix ensures a bbox height of at least fontsize.
- **Changed** for ListBox and ComboBox widgets, the attribute list of selectable values has been renamed to *Widget.choice_values*.
- **Changed** when adding widgets, any missing of the *PDF Base 14 Fonts* is automatically added to the PDF. Widget text fonts can now also be chosen from existing widget fonts. Any specified field values are now honored and lead to a field with a preset value.
- **Added** `Annot.updateWidget()` which allows changing existing form fields – including the field value.

15.36 Changes in Version 1.13.11

While the preceeding patch subversions only contained various fixes, this version again introduces major new features:

- **Added** basic support for PDF widget annotations. You can now add PDF form fields of types Text, CheckBox, ListBox and ComboBox. Where necessary, the PDF is tranformed to a Form PDF with the first added widget.
- **Fixed** issues #176 ("wrong file embedding"), #177 ("segment fault when invoking page.getText()")and #179 ("Segmentation fault using page.getLinks() on encrypted PDF").

15.37 Changes in Version 1.13.7

- **Added** support of variable page sizes for reflowable documents (e-books, HTML, etc.): new parameters `rect` and `fontsize` in *Document* creation (open), and as a separate method *Document.layout()*.
- **Added** *Annot* creation of many annotations types: sticky notes, free text, circle, rectangle, line, polygon, polyline and text markers.
- **Added** support of annotation transparency (*Annot.opacity*, *Annot.setOpacity()*).
- **Changed** *Annot.vertices*: point coordinates are now grouped as pairs of floats (no longer as separate floats).
- **Changed** annotation colors dictionary: the two keys are now named `"stroke"` (formerly `"common"`) and `"fill"`.
- **Added** `Document.isDirty` which is `True` if a PDF has been changed in this session. Reset to `False` on each *Document.save()* or *Document.write()*.

15.38 Changes in Version 1.13.6

- Fix #173: for memory-resident documents, ensure the stream object will not be garbage-collected by Python before document is closed.

15.39 Changes in Version 1.13.5

- New low-level method *Page._setContents()* defines an object given by its *xref* to serve as the *contents* object.
- Changed and extended PDF form field support: the attribute `widget_text` has been renamed to `Annot.widget_value`. Values of all form field types (except signatures) are now supported. A new attribute `Annot.widget_choices` contains the selectable values of listboxes and comboboxes. All these attributes now contain `None` if no value is present.

15.40 Changes in Version 1.13.4

- *Document.convertToPDF()* now supports page ranges, reverted page sequences and page rotation. If the document already is a PDF, an exception is raised.
- Fixed a bug (introduced with v1.13.0) that prevented *Page.insertImage()* for transparent images.

15.41 Changes in Version 1.13.3

Introduces a way to convert **any MuPDF supported document** to a PDF. If you ever wanted PDF versions of your XPS, EPUB, CBZ or FB2 files – here is a way to do this.

- *Document.convertToPDF()* returns a Python `bytes` object in PDF format. Can be opened like normal in PyMuPDF, or be written to disk with the `".pdf"` extension.

15.42 Changes in Version 1.13.2

The major enhancement is PDF form field support. Form fields are annotations of type `(19, 'Widget')`. There is a new document method to check whether a PDF is a form. The *Annot* class has new properties describing field details.

- *Document.isFormPDF* is true if object type `/AcroForm` and at least one form field exists.
- `Annot.widget_type`, `Annot.widget_text` and `Annot.widget_name` contain the details of a form field (i.e. a "Widget" annotation).

15.43 Changes in Version 1.13.1

- *TextPage.extractDICT()* is a new method to extract the contents of a document page (text and images). All document types are supported as with the other *TextPage* `extract*()` methods. The

returned object is a dictionary of nested lists and other dictionaries, and **exactly equal** to the JSON-deserialization of the old *TextPage.extractJSON()*. The difference is that the result is created directly – no JSON module is used. Because the user needs no JSON module to interpet the information, it should be easier to use, and also have a better performance, because it contains images in their original **binary format** – they need not be base64-decoded.

- *Page.getText()* correspondingly supports the new parameter value `"dict"` to invoke the above method.
- *TextPage.extractJSON()* (resp. `Page.getText("json")`) is still supported for convenience, but its use is expected to decline.

15.44 Changes in Version 1.13.0

This version is based on MuPDF v1.13.0. This release is "primarily a bug fix release".

In PyMuPDF, we are also doing some bug fixes while introducing minor enhancements. There only very minimal changes to the user's API.

- *Document* construction is more flexible: the new `filetype` parameter allows setting the document type. If specified, any extension in the filename will be ignored. More completely addresses issue #156[100]. As part of this, the documentation has been reworked.
- **Changes to *Pixmap* constructors:**
  - Colorspace conversion no longer allows dropping the alpha channel: source and target **alpha will now always be the same**. We have seen exceptions and even interpreter crashes when using `alpha = 0`.
  - As a replacement, the simple pixmap copy lets you choose the target alpha.
- *Document.save()* again offers the full garbage collection range 0 thru 4. Because of a bug in *xref* maintenance, we had to temporarily enforce `garbage > 1`. Finally resolves issue #148[101].
- *Document.save()* now offers to "prettify" PDF source via an additional argument.
- *Page.insertImage()* has the additional `stream` -parameter, specifying a memory area holding an image.
- Issue with garbled PNGs on Linux systems has been resolved ("Problem writing PNG" #133)[102].

15.45 Changes in Version 1.12.4

This is an extension of 1.12.3.

- Fix of issue #147[103]: methods `Document.getPageFontlist()` and `Document.getPageImagelist()` now also show fonts and images contained in *resources* nested via "Form XObjects".
- Temporary fix of issue #148[104]: Saving to new PDF files will now automatically use `garbage = 2` if a lower value is given. Final fix is to be expected with MuPDF's next version. At that point we will remove this circumvention.
- Preventive fix of illegally using stencil / image mask pixmaps in some methods.

[100] https://github.com/rk700/PyMuPDF/issues/156
[101] https://github.com/rk700/PyMuPDF/issues/148
[102] https://github.com/rk700/PyMuPDF/issues/133
[103] https://github.com/rk700/PyMuPDF/issues/147
[104] https://github.com/rk700/PyMuPDF/issues/148

- Method `Document.getPageFontlist()` now includes the encoding name for each font in the list.
- Method `Document.getPageImagelist()` now includes the decode method name for each image in the list.

15.46 Changes in Version 1.12.3

This is an extension of 1.12.2.

- Many functions now return `None` instead of `0`, if the result has no other meaning than just indicating successful execution (*Document.close()*, *Document.save()*, *Document.select()*, *Pixmap.writePNG()* and many others).

15.47 Changes in Version 1.12.2

This is an extension of 1.12.1.

- Method *Page.showPDFpage()* now accepts the new `clip` argument. This specifies an area of the source page to which the display should be restricted.
- New *Page.CropBox* and *Page.MediaBox* have been included for convenience.

15.48 Changes in Version 1.12.1

This is an extension of version 1.12.0.

- New method *Page.showPDFpage()* displays another's PDF page. This is a **vector** image and therefore remains precise across zooming. Both involved documents must be PDF.
- New method *Page.getSVGimage()* creates an SVG image from the page. In contrast to the raster image of a pixmap, this is a vector image format. The return is a unicode text string, which can be saved in a `.svg` file.
- Method *Page.getTextBlocks()* now accepts an additional bool parameter "images". If set to true (default is false), image blocks (metadata only) are included in the produced list and thus allow detecting areas with rendered images.
- Minor bug fixes.
- "text" result of *Page.getText()* concatenates all lines within a block using a single space character. MuPDF's original uses "\n" instead, producing a rather ragged output.
- New properties of *Page* objects *Page.MediaBoxSize* and *Page.CropBoxPosition* provide more information about a page's dimensions. For non-PDF files (and for most PDF files, too) these will be equal to `Page.rect.bottom_right`, resp. `Page.rect.top_left`. For example, class *Shape* makes use of them to correctly position its items.

15.49 Changes in Version 1.12.0

This version is based on and requires MuPDF v1.12.0. The new MuPDF version contains quite a number of changes – most of them around text extraction. Some of the changes impact the programmer's API.

- `Outline.saveText()` and `Outline.saveXML()` have been deleted without replacement. You probably haven't used them much anyway. But if you are looking for a replacement: the output of *Document.getToC()* can easily be used to produce something equivalent.
- Class `TextSheet` does no longer exist.
- Text "spans" (one of the hierarchy levels of *TextPage*) no longer contain positioning information (i.e. no "bbox" key). Instead, spans now provide the font information for its text. This impacts our JSON output variant.
- HTML output has improved very much: it now creates valid documents which can be displayed by browsers to produce a similar view as the original document.
- There is a new output format XHTML, which provides text and images in a browser-readable format. The difference to HTML output is, that no effort is made to reproduce the original layout.
- All output formats of *Page.getText()* now support creating complete, valid documents, by wrapping them with appropriate header and trailer information. If you are interested in using the HTML output, please make sure to read *Controlling Quality of HTML Output*.
- To support finding text positions, we have added special methods that don't need detours like *TextPage.extractJSON()* or *TextPage.extractXML()*: use *Page.getTextBlocks()* or resp. *Page.getTextWords()* to create lists of text blocks or resp. words, which are accompanied by their rectangles. This should be much faster than the standard text extraction methods and also avoids using additional packages for interpreting their output.

15.50 Changes in Version 1.11.2

This is an extension of v1.11.1.

- New *Page.insertFont()* creates a PDF `/Font` object and returns its object number.
- New *Document.extractFont()* extracts the content of an embedded font given its object number.
- Methods `*FontList(...)` items no longer contain the PDF generation number. This value never had any significance. Instead, the font file extension is included (e.g. "pfa" for a "PostScript Font for ASCII"), which is more valuable information.
- Fonts other than "simple fonts" (Type1) are now also supported.
- New options to change *Pixmap* size:
  - Method *Pixmap.shrink()* reduces the pixmap proportionally in place.
  - A new *Pixmap* copy constructor allows scaling via setting target width and height.

15.51 Changes in Version 1.11.1

This is an extension of v1.11.0.

- New class `Shape`. It facilitates and extends the creation of image shapes on PDF pages. It contains multiple methods for creating elementary shapes like lines, rectangles or circles, which can be combined into more complex ones and be given common properties like line width or colors. Combined shapes are handled as a unit and e.g. be "morphed" together. The class can accumulate multiple complex shapes and put them all in the page's foreground or background – thus also reducing the number of updates to the page's *contents* object.
- All `Page` draw methods now use the new `Shape` class.

- Text insertion methods `insertText()` and `insertTextBox()` now support morphing in addition to text rotation. They have become part of the `Shape` class and thus allow text to be freely combined with graphics.
- A new `Pixmap` constructor allows creating pixmap copies with an added alpha channel. A new method also allows directly manipulating alpha values.
- Binary algebraic operations with geometry objects (matrices, rectangles and points) now generally also support lists or tuples as the second operand. You can add a tuple `(x, y)` of numbers to a *Point*. In this context, such sequences are called "*point_like*" (resp. *matrix_like*, *rect_like*).
- Geometry objects now fully support in-place operators. For example, `p /= m` replaces point p with `p * 1/m` for a number, or `p * ~m` for a *matrix_like* object `m`. Similarly, if `r` is a rectangle, then `r |= (3, 4)` is the new rectangle that also includes `fitz.Point(3, 4)`, and `r &= (1, 2, 3, 4)` is its intersection with `fitz.Rect(1, 2, 3, 4)`.

15.52 Changes in Version 1.11.0

This version is based on and requires MuPDF v1.11.

Though MuPDF has declared it as being mostly a bug fix version, one major new feature is indeed contained: support of embedded files – also called portfolios or collections. We have extended PyMuPDF functionality to embrace this up to an extent just a little beyond the `mutool` utility as follows.

- The `Document` class now support embedded files with several new methods and one new property:
  - `embeddedFileInfo()` returns metadata information about an entry in the list of embedded files. This is more than `mutool` currently provides: it shows all the information that was used to embed the file (not just the entry's name).
  - `embeddedFileGet()` retrieves the (decompressed) content of an entry into a `bytes` buffer.
  - `embeddedFileAdd(...)` inserts new content into the PDF portfolio. We (in contrast to `mutool`) **restrict** this to entries with a **new name** (no duplicate names allowed).
  - `embeddedFileDel(...)` deletes an entry from the portfolio (function not offered in MuPDF).
  - `embeddedFileSetInfo()` – changes filename or description of an embedded file.
  - `embeddedFileCount` – contains the number of embedded files.
- Several enhancements deal with streamlining geometry objects. These are not connected to the new MuPDF version and most of them are also reflected in PyMuPDF v1.10.0. Among them are new properties to identify the corners of rectangles by name (e.g. `Rect.bottom_right`) and new methods to deal with set-theoretic questions like `Rect.contains(x)` or `IRect.intersects(x)`. Special effort focussed on supporting more "Pythonic" language constructs: `if x in rect ...` is equivalent to `rect.contains(x)`.
- The *Rect* chapter now has more background on empty amd infinite rectangles and how we handle them. The handling itself was also updated for more consistency in this area.
- We have started basic support for **generation** of PDF content:
  - `Document.insertPage()` adds a new page into a PDF, optionally containing some text.
  - `Page.insertImage()` places a new image on a PDF page.
  - `Page.insertText()` puts new text on an existing page
- For **FileAttachment** annotations, content and name of the attached file can extracted and changed.

15.53 Changes in Version 1.10.0

15.53.1 MuPDF v1.10 Impact

MuPDF version 1.10 has a significant impact on our bindings. Some of the changes also affect the API – in other words, **you** as a PyMuPDF user.

- Link destination information has been reduced. Several properties of the `linkDest` class no longer contain valuable information. In fact, this class as a whole has been deleted from MuPDF's library and we in PyMuPDF only maintain it to provide compatibilty to existing code.
- In an effort to minimize memory requirements, several improvements have been built into MuPDF v1.10:
  - A new `config.h` file can be used to de-select unwanted features in the C base code. Using this feature we have been able to reduce the size of our binary `_fitz.o` / `_fitz.pyd` by about 50% (from 9 MB to 4.5 MB). When UPX-ing this, the size goes even further down to a very handy 2.3 MB.
  - The alpha (transparency) channel for pixmaps is now optional. Letting alpha default to `False` significantly reduces pixmap sizes (by 20% – CMYK, 25% – RGB, 50% – GRAY). Many `Pixmap` constructors therefore now accept an `alpha` boolean to control inclusion of this channel. Other pixmap constructors (e.g. those for file and image input) create pixmaps with no alpha altogether. On the downside, save methods for pixmaps no longer accept a `savealpha` option: this channel will always be saved when present. To minimize code breaks, we have left this parameter in the call patterns – it will just be ignored.
- `DisplayList` and `TextPage` class constructors now **require the mediabox** of the page they are referring to (i.e. the `page.bound()` rectangle). There is no way to construct this information from other sources, therefore a source code change cannot be avoided in these cases. We assume however, that not many users are actually employing these rather low level classes explixitely. So the impact of that change should be minor.

15.53.2 Other Changes compared to Version 1.9.3

- The new *Document* method `write()` writes an opened PDF to memory (as opposed to a file, like `save()` does).
- An annotation can now be scaled and moved around on its page. This is done by modifying its rectangle.
- Annotations can now be deleted. *Page* contains the new method `deleteAnnot()`.
- Various annotation attributes can now be modified, e.g. content, dates, title (= author), border, colors.
- Method `Document.insertPDF()` now also copies annotations of source pages.
- The `Pages` class has been deleted. As documents can now be accessed with page numbers as indices (like `doc[n] = doc.loadPage(n)`), and document object can be used as iterators, the benefit of this class was too low to maintain it. See the following comments.
- `loadPage(n)` / `doc[n]` now accept arbitrary integers to specify a page number, as long as `n < pageCount`. So, e.g. `doc[-500]` is always valid and will load page `(-500) % pageCount`.
- A document can now also be used as an iterator like this: `for page in doc: ...<do something with "page"> ...`. This will yield all pages of `doc` as `page`.

- The *Pixmap* method `getSize()` has been replaced with property `size`. As before `Pixmap.size == len(Pixmap)` is true.
- In response to transparency (alpha) being optional, several new parameters and properties have been added to *Pixmap* and *Colorspace* classes to support determining their characteristics.
- The *Page* class now contains new properties `firstAnnot` and `firstLink` to provide starting points to the respective class chains, where `firstLink` is just a mnemonic synonym to method `loadLinks()` which continues to exist. Similarly, the new property `rect` is a synonym for method `bound()`, which also continues to exist.
- *Pixmap* methods `samplesRGB()` and `samplesAlpha()` have been deleted because pixmaps can now be created without transparency.
- *Rect* now has a property `irect` which is a synonym of method `round()`. Likewise, *IRect* now has property `rect` to deliver a *Rect* which has the same coordinates as floats values.
- Document has the new method `searchPageFor()` to search for a text string. It works exactly like the corresponding `Page.searchFor()` with page number as additional parameter.

15.54 Changes in Version 1.9.3

This version is also based on MuPDF v1.9a. Changes compared to version 1.9.2:

- As a major enhancement, annotations are now supported in a similar way as links. Annotations can be displayed (as pixmaps) and their properties can be accessed.
- In addition to the document `select()` method, some simpler methods can now be used to manipulate a PDF:
  - `copyPage()` copies a page within a document.
  - `movePage()` is similar, but deletes the original.
  - `deletePage()` deletes a page
  - `deletePageRange()` deletes a page range
- `rotation` or `setRotation()` access or change a PDF page's rotation, respectively.
- Available but undocumented before, *IRect*, *Rect*, *Point* and *Matrix* support the `len()` method and their coordinate properties can be accessed via indices, e.g. `IRect.x1 == IRect[2]`.
- For convenience, documents now support simple indexing: `doc.loadPage(n) == doc[n]`. The index may however be in range `-pageCount < n < pageCount`, such that `doc[-1]` is the last page of the document.

15.55 Changes in Version 1.9.2

This version is also based on MuPDF v1.9a. Changes compared to version 1.9.1:

- `fitz.open()` (no parameters) creates a new empty **PDF** document, i.e. if saved afterwards, it must be given a `.pdf` extension.
- *Document* now accepts all of the following formats (`Document` and `open` are synonyms):
  - `open()`,
  - `open(filename)` (equivalent to `open(filename, None)`),

  - `open(filetype, area)` (equivalent to `open(filetype, stream = area)`).

  Type of memory area `stream` may be `bytes` or `bytearray`. Thus, e.g. `area = open("file.pdf", "rb").read()` may be used directly (without first converting it to bytearray).

- New method `Document.insertPDF()` (PDFs only) inserts a range of pages from another PDF.
- `Document` objects doc now support the `len()` function: `len(doc) == doc.pageCount`.
- New method `Document.getPageImageList()` creates a list of images used on a page.
- New method `Document.getPageFontList()` creates a list of fonts referenced by a page.
- New pixmap constructor `fitz.Pixmap(doc, xref)` creates a pixmap based on an opened PDF document and an *`xref`* number of the image.
- New pixmap constructor `fitz.Pixmap(cspace, spix)` creates a pixmap as a copy of another one `spix` with the colorspace converted to `cspace`. This works for all colorspace combinations.
- Pixmap constructor `fitz.Pixmap(colorspace, width, height, samples)` now allows `samples` to also be `bytes`, not only `bytearray`.

15.56 Changes in Version 1.9.1

This version of PyMuPDF is based on MuPDF library source code version 1.9a published on April 21, 2016.

Please have a look at MuPDF's website to see which changes and enhancements are contained herein.

Changes in version 1.9.1 compared to version 1.8.0 are the following:

- New methods `getRectArea()` for both `fitz.Rect` and `fitz.IRect`
- Pixmaps can now be created directly from files using the new constructor `fitz.Pixmap(filename)`.
- The Pixmap constructor `fitz.Pixmap(image)` has been extended accordingly.
- `fitz.Rect` can now be created with all possible combinations of points and coordinates.
- PyMuPDF classes and methods now all contain __doc__ strings, most of them created by SWIG automatically. While the PyMuPDF documentation certainly is more detailed, this feature should help a lot when programming in Python-aware IDEs.
- A new document method of `getPermits()` returns the permissions associated with the current access to the document (print, edit, annotate, copy), as a Python dictionary.
- The identity matrix `fitz.Identity` is now **immutable**.
- The new document method `select(list)` removes all pages from a document that are not contained in the list. Pages can also be duplicated and re-arranged.
- Various improvements and new members in our demo and examples collections. Perhaps most prominently: `PDF_display` now supports scrolling with the mouse wheel, and there is a new example program `wxTableExtract` which allows to graphically identify and extract table data in documents.
- `fitz.open()` is now an alias of `fitz.Document()`.
- New pixmap method `getPNGData()` which will return a bytearray formatted as a PNG image of the pixmap.
- New pixmap method `samplesRGB()` providing a `samples` version with alpha bytes stripped off (RGB colorspaces only).
- New pixmap method `samplesAlpha()` providing the alpha bytes only of the `samples` area.

- New iterator `fitz.Pages(doc)` over a document's set of pages.
- New matrix methods `invert()` (calculate inverted matrix), `concat()` (calculate matrix product), `preTranslate()` (perform a shift operation).
- New `IRect` methods `intersect()` (intersection with another rectangle), `translate()` (perform a shift operation).
- New `Rect` methods `intersect()` (intersection with another rectangle), `transform()` (transformation with a matrix), `includePoint()` (enlarge rectangle to also contain a point), `includeRect()` (enlarge rectangle to also contain another one).
- Documented `Point.transform()` (transform a point with a matrix).
- `Matrix`, `IRect`, `Rect` and `Point` classes now support compact, algebraic formulations for manipulating such objects.
- Incremental saves for changes are possible now using the call pattern `doc.save(doc.name, incremental=True)`.
- A PDF's metadata can now be deleted, set or changed by document method `setMetadata()`. Supports incremental saves.
- A PDF's bookmarks (or table of contents) can now be deleted, set or changed with the entries of a list using document method `setToC(list)`. Supports incremental saves.

INDEX