PyMuPDF Documentation

Release 1.12.2

Jorj X. McKie

Jan 09, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

PyMuPDF is a Python binding for MuPDF - "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 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 (first v1.7a, up to v1.12.0 as of this writing).

PyMuPDF runs and has been tested on Mac, Linux, Windows XP SP2 and up, Python 2.7 through Python 3.6 (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. Because we rely on MuPDF's C library, installation consists of two separate steps for all platforms except for MS Windows:

1. Installation of MuPDF: this involves downloading the source from their website and then compiling it on your machine.
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 the Windows platform we have however combined these steps and offer binaries, available in ZIP and wheel formats. This installation material is contained in a separate GitHub repository and obsoletes all other download and generation work. You only need to choose which Python version and bitness you want and then download the respective zip or wheel file (less than 3 MB).

For installation details check out the respective chapter.

We also are registered on PyPI.

There exist several demo and example 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).

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, if you want to create a commercial product with PyMuPDF.

1.3 Covered Version

This documentation covers PyMuPDF 1.12.2 features as of **2018-01-09, 09:41:34**.

CHAPTER

TWO

INSTALLATION

Installation generally encompasses downloading and generating PyMuPDF and MuPDF from sources.

This process consists of three steps described below under *Option 1: Install from Sources*.

However, if your operating system is MS Windows, you can perform a binary setup, detailed out under *Option 2: Install from Binaries*. This process is **much faster** and requires the download of only one 3 MB file (either `.zip` or `.whl`) - no compiler, no Visual Studio, no download of MuPDF, even no download of PyMuPDF.

2.1 Option 1: Install from Sources

2.1.1 Step 1: Download PyMuPDF

Download this repository and unzip / decompress it. This will give you a folder, let us call it `PyFitz`.

2.1.2 Step 2: Download and Generate MuPDF

Download `mupdf-x.xx-source.tar.gz` from http://mupdf.com/downloads and unzip / decompress it. Call the resulting folder `mupdf`. The latest MuPDF **development sources** are available on https://github.com/ArtifexSoftware/mupdf - this is **not** what you want here.

Make sure you download the (sub-) version for which PyMuPDF has stated its compatibility. The various Linux flavors usually have their own specific ways to support download of packages which we cannot cover here. Do not hesitate posting issues to our web site or sending an e-mail to the authors for getting support.

Put it inside `PyFitz` as a subdirectory for keeping everything in one place.

Controlling the Binary File Size:

Since version 1.9, MuPDF includes support for many dozens of additional, so-called NOTO ("no TOFU") fonts for all sorts of alphabets from all over the world like Chinese, Japanese, Corean, Kyrillic, Indonesian, Chinese etc. If you accept MuPDF's standard here, the resulting binary for PyMuPDF will be very big and easily approach or exceed 20 MB. The features actually needed by PyMuPDF in contrast only represent a fraction of this size: no more than 5 MB currently.

To cut off unneeded stuff from your MuPDF version, modify file `/include/mupdf/config.h` as follows:

```
#ifndef FZ_CONFIG_H

#define FZ_CONFIG_H

/*
Enable the following for spot (and hence overprint/overprint
simulation) capable rendering. This forces FZ_PLOTTERS_N on.
*/
#define FZ_ENABLE_SPOT_RENDERING
```

```
/*
Choose which plotters we need.
By default we build all the plotters in. To avoid building
plotters in that aren't needed, define the unwanted
FZ_PLOTTERS_... define to 0.
*/
/* #define FZ_PLOTTERS_G 1 */
/* #define FZ_PLOTTERS_RGB 1 */
/* #define FZ_PLOTTERS_CMYK 1 */
/* #define FZ_PLOTTERS_N 1 */

/*
Choose which document agents to include.
By default all but GPRF are enabled. To avoid building unwanted
ones, define FZ_ENABLE_... to 0.
*/
/* #define FZ_ENABLE_PDF 1 */
/* #define FZ_ENABLE_XPS 1 */
/* #define FZ_ENABLE_SVG 1 */
/* #define FZ_ENABLE_CBZ 1 */
/* #define FZ_ENABLE_IMG 1 */
/* #define FZ_ENABLE_TIFF 1 */
/* #define FZ_ENABLE_HTML 1 */
/* #define FZ_ENABLE_EPUB 1 */
/* #define FZ_ENABLE_GPRF 1 */

/*
Choose whether to enable JPEG2000 decoding.
By default, it is enabled, but due to frequent security
issues with the third party libraries we support disabling
it with this flag.
*/
/* #define FZ_ENABLE_JPX 1 */

/*
Choose whether to enable JavaScript.
By default JavaScript is enabled both for mutool and PDF interactivity.
*/
/* #define FZ_ENABLE_JS 1 */

/*
Choose which fonts to include.
By default we include the base 14 PDF fonts,
DroidSansFallback from Android for CJK, and
Charis SIL from SIL for epub/html.
Enable the following defines to AVOID including
unwanted fonts.
*/
/* To avoid all noto fonts except CJK, enable: */
#define TOFU // PyMuPDF

/* To skip the CJK font, enable: (this implicitly enables TOFU_CJK_EXT and TOFU_CJK_LANG) */
#define TOFU_CJK // PyMuPDF

/* To skip CJK Extension A, enable: (this implicitly enables TOFU_CJK_LANG) */
#define TOFU_CJK_EXT // PyMuPDF

/* To skip CJK language specific fonts, enable: */
#define TOFU_CJK_LANG // PyMuPDF

/* To skip the Emoji font, enable: */
#define TOFU_EMOJI // PyMuPDF
```

```
/* To skip the ancient/historic scripts, enable: */
#define TOFU_HISTORIC // PyMuPDF

/* To skip the symbol font, enable: */
#define TOFU_SYMBOL // PyMuPDF

/* To skip the SIL fonts, enable: */
#define TOFU_SIL // PyMuPDF

/* To skip the ICC profiles, enable: */
#define NO_ICC // PyMuPDF

/* To skip the Base14 fonts, enable: */
/* #define TOFU_BASE14 */
/* (You probably really don't want to do that except for measurement purposes!) */

/* ---------- DO NOT EDIT ANYTHING UNDER THIS LINE ---------- */

... ... ...

#endif /* FZ_CONFIG_H */
```

The above choice should bring down your binary file size to around 5 MB or less, depending on your bitness.

Generate MuPDF now.

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

Consult additional installation hints on PyMuPDF's main page on Github. Among other things you will find a Wiki pages with details on building the Windows binaries or user provided installation experiences.

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`.

2.2 Option 2: Install from Binaries

This installation option is available for the lucky **MS Windows users only**. All versions of Windows (XP SP2 and up) and Python (2.7 and up) are supported with either 32bit or 64bit at your choice.

2.2.1 Step 1: Download Binary

You do not need the complete repository PyMuPDF-optional-material.

Either

- issue `pip install PyMuPDF [--upgrade]` and **you are done**,

or

- download the ZIP or WHL file you need and **read on**.

2.2.2 Step 2: Install PyMuPDF

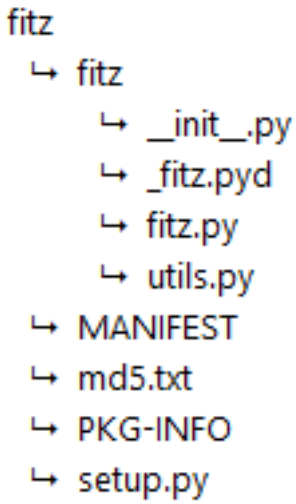
The next steps are of interest if you have special needs. For example: if you do not want to use `pip`, you can do a ZIP-file-based installation. Or you may want to install a wheel that does not target your standard Python, or is a pre-release build, etc.

- If you have downloaded a wheel, install it via `pip install PyMuPDF-<...>.whl [--upgrade]` and **you are done**.
- If you have downloaded a zip, unzip it to e.g. your `Desktop` and open a command prompt at the unzipped folder's directory, which contains `setup.py`. Enter `python setup.py install` (or `py setup.py install` if you have the Python launcher).

2.2.3 MD5 Checksums

Binary download setup scripts in ZIP format contain an integrity check based on MD5 check sums.

The directory structure of each zip file `pymupdf-<...>.zip` is as follows:



During setup, the MD5 check sum of the four installation files `__init__.py`, `_fitz.pyd`, `utils.py` and `fitz.py` is being calculated and compared against a pre-calculated value in file `md5.txt`. In case of a mismatch the error message

```
md5 mismatch:  probable download error
```

is issued and setup is cancelled. In this case, please check your download for any problems.

If you downloaded a wheel, integrity checks are done by `pip`.

2.2.4 Targeting Parallel Python Installations

Setup scripts for ZIP binary install support the Python launcher `py.exe` introduced with version 3.3.

They contain **shebang lines** that specify the intended Python version, and additional checks for detecting error situations.

This can be used to target the right Python version if you have several installed in parallel (and of course the Python launcher, too). Use the following statement to set up PyMuPDF correctly:

```
py setup.py install
```

The shebang line of `setup.py` will be interpreted by `py.exe` to automatically find the right Python, and the internal checks will make sure that version and bitness are what they sould be.

When using wheels, configuration conflict detection is done by `pip`.

2.3 Using UPX

No matter which option you chose, your PyMuPDF installation will end up with four files: `__init__.py`, `fitz.py`, `utils.py` and the binary file `_fitz.xxx` in the `site-packages` directory. The extension of the binary will be `.pyd` on Windows and `.so` on other platforms.

Depending on your OS, your compiler and your font support choice (see above), this binary can be quite large and range from 5 MB to 20 MB. You can reduce this by applying the compression utility UPX to it, which probably also exists for your operating system. UPX will reduce the size of `_fitz.xxx` by more than 50%. You will end up with 2.5 MB to 9 MB without impacting functionality nor execution speed.

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[1]. Nevertheless we will only talk about PDF files for the sake of brevity. 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:

```
>>> import fitz
```

You can check your version by printing the docstring:

```
>>> print (fitz.__doc__)
PyMuPDF 1.9.1: Python bindings for the MuPDF 1.9a library,
built on 2016-07-01 13:06:02
```

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 a *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.

3.3 Some Document Methods and Attributes

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

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

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 contain data in every single case. For details of their meanings and formats consult the PDF manuals, e.g. *Adobe PDF Reference 1.7*. Further information can also be found in chapter *Document*. The meta data fields are strings (or `None`) if not otherwise indicated. Be aware that not all of them necessarily contain meaningful data.

Key	Value
producer	producer (producing software)
format	PDF format, e.g. 'PDF-1.4'
encryption	encryption method used
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 of PDF version 1.4 or later may also contain so-called *"metadata streams"*. Information in such streams is coded in XML. PyMuPDF deliberately contains no XML components, so we do not directly support access to information contained therein. But you can extract the stream as a whole, inspect or modify it using a package like lxml and then store the result back into the PDF. If you want, you can also delete these data altogether.

3.5 Working with Outlines

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

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

This will return a Python list of lists `[[lvl, title, page, ...], ...]`.

`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.

3.6 Working with Pages

Tasks that can be performed with a *Page* are at the core of MuPDF's functionality.

- You can render a page into an image, optionally zooming, rotating, shifting or shearing it.
- You can extract a page's text or search for text strings.

First, a page object must be created:

```
>>> page = doc.loadPage(n)        # represents page n of the document (0-based)
>>> page = doc[n]                 # short form
```

`n` may be any integer less than the total number of pages of the document. `doc[-1]` is the last page, like with Python lists.

Some typical uses of *Page*s follow:

3.6.1 Inspecting the Links of a Page

Here is how to get all links and their types:

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

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

3.6.2 Rendering a Page

This example creates an image out of a page's content:

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

Now `pix` is a *Pixmap* object that contains an RGBA image of the page, ready to be used. This method offers lots of variations for controlling the image: resolution, colorspace, transparency, rotation, mirroring, shifting, shearing, etc.

3.6.3 Saving the Page Image in a File

We can simply store the image in a PNG file:

```
>>> pix.writePNG("test.png")
```

3.6.4 Displaying the Image in Dialog Managers

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

wxPython:

```
>>> bitmap = wx.BitmapFromBufferRGBA(pix.width, pix.height, pix.samples)
```

Tkinter:

```
>>> # the following requires: "from PIL import Image, ImageTk"
>>> img = Image.frombytes("RGBA", [pix.width, pix.height], pix.samples)
>>> photo = ImageTk.PhotoImage(img)
```

Now, `photo` can be used as an image in TK.

3.6.5 Extracting Text

We can also extract all text of a page in one chunk of string:

```
>>> text = page.getText(type)
```

Use one of the following strings for `type`:

- `"text"`: (default) plain text with line breaks. No formatting, no text position details.
- `"html"`: creates a full visual version of the page including any images, which can be displayed in browsers.
- `"json"`: same information level as HTML. Use a JSON module to interpret.
- `"xhtml"`: text information level as the TEXT version, but includes images and can also be displayed in browsers.

- `"xml"`: contains no images, but full position and font information about 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 Text

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

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

The variable `areas` will contain a list of up to 16 *Rect*angles, each of which surrounds one occurrence of the string "mupdf" (case insensitive). You could use this information to e.g. highlight those areas or create a cross reference of the document.

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

3.7 PDF Maintenance

Since version 1.9, PyMuPDF provides several options to modify PDF documents (only).

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

Apart from your changes, there are less obvious ways for a PDF to becoming "modified":

- During open, integrity checks are used to determine the health of the PDF structure. Any errors will be corrected as far as possible to present a repaired document in memory for further processing. If this is the case, the document is regarded as being modified.
- After a document has been decrypted, the document in memory has changed and also counts as being modified.

In these two cases, *Document.save()* will store a repaired and / or decrypted version, and saving **must occur to a new file**.

The following describe some more intentional ways to manipulate PDF documents. This description is by no means exhaustive: 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 page tree of a PDF:

Methods *Document.deletePage()* and *Document.deletePageRange()* delete pages.

Methods *Document.copyPage()* and *Document.movePage()* copy or move a page to another location within the document.

Document.insertPage() and *Document.newPage()* insert pages.

Method *Document.select()* shrinks a document down to selected pages. It accepts a sequence of integers as argument. These integers must be in range `0 <= i < pageCount`. When executed, all pages **missing** in this list will be deleted. Only pages that do occur will remain - **in the sequence specified and as many times (!) as specified**.

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 certain text, reverse their sequence, ... whatever you may think of.

The saved new document will contain all still valid links, annotations and bookmarks.

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()* inserts pages from another PDF at a specified place of the current one. Here is a simple **joiner** example (`doc1` and `doc2` being openend PDFs):

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

Here is how to split `doc1`. This creates a new document of its first and last 10 pages (could also be done using *Document.select()*):

```
>>> doc2 = fitz.open()                  # new empty PDF
>>> doc2.insertPDF(doc1, to_page = 9)
>>> doc2.insertPDF(doc1, from_page = len(doc1) - 10)
>>> doc2.save(...)
```

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

3.7.3 Saving

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

Since MuPDF 1.9, you can write changes back to the original PDF by specifying `incremental = True`. This process is (usually) **extremely fast**, since changes are **appended to the original file** without rewriting it.

Document.save() supports all options of MuPDF's command line utility `mutool clean`, see the following table (corresponding `mutool clean` option is indicated as "mco").

Option	mco	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	s	clean 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
incremental = 1	n/a	append changes to the original

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

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 Example: Dynamically Cleaning up Corrupt PDF Documents

This shows a potential use of PyMuPDF with another Python PDF library (pdfrw).

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

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

#---------------------------------------
# 'tolerant' PDF reader
#---------------------------------------
def reader(fname):
    ifile = open(fname, "rb")
    idata = ifile.read()                    # put in memory
    ifile.close()
    ibuffer = BytesIO(idata)                # convert to stream
    try:
        return PdfReader(ibuffer)           # let us try
    except:                                 # problem! heal it with PyMuPDF
        doc = fitz.open("pdf", idata)       # open and save a corrected
        c = doc.write(garbage = 4)          # version in memory
        doc.close()
        doc = idata = None                  # free storage
        ibuffer = BytesIO(c)                # convert to stream
        return PdfReader(ibuffer)           # let pdfrw retry
#---------------------------------------
# Main program
#---------------------------------------
pdf = reader("pymupdf.pdf")
print pdf.Info
# do further processing
```

With the command line utility `pdftk` (available for Windows only) a similar result can be achieved, see here. However, you must invoke it as a separate process via `subprocess.Popen`, using stdin and stdout as communication vehicles.

3.10 Further Reading

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

CHAPTER

FOUR

CLASSES

4.1 Annot

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."

This class supports accessing such annotations - not only for PDF files, but for all MuPDF supported document types. However, only a few methods and properties apply to non-PDF documents.

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.getPixmap()	image of the annotation as a pixmap
Annot.setInfo()	PDF only: change metadata of an annotation
Annot.setBorder()	PDF only: changes the border of an annotation
Annot.setFlags()	PDF only: changes the flags of an annotation
Annot.setRect()	PDF only: changes the rectangle of an annotation
Annot.setColors()	PDF only: changes the colors of an annotation
Annot.updateImage()	PDF only: applies border and color values to shown image
Annot.fileInfo()	PDF only: returns attached file information
Annot.fileGet()	PDF only: returns attached file content
Annot.fileUpd()	PDF only: sets attached file new content
Annot.border	PDF only: border details
Annot.colors	PDF only: border / background and fill colors
Annot.flags	PDF only: annotation flags
Annot.info	PDF only: various information
Annot.lineEnds	PDF only: start / end appearance of line-type annotations
Annot.next	link to the next annotation
Annot.parent	page object of the annotation
Annot.rect	rectangle containing the annotation
Annot.type	PDF only: type of the annotation
Annot.vertices	PDF only: point coordinates of Polygons, PolyLines, etc.

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 is 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 `unicode`, `bytes`, or strings. If `bytes` values in Python 3 they will be treated as being UTF8 encoded.

`setRect(rect)`

Changes 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*) – the new rectangle of the annotation. This could e.g. be a rectangle `rect = Annot.rect * M` with a suitable *Matrix* M (only scaling and translating will yield the expected effect).

`setBorder(value)`

PDF only: Change border width and dashing properties. Any other border properties will be deleted.

Parameters `value` (*float or dict*) – a number or a dictionary specifying the desired border properties. If a dictionary, its `width` and `dashes` keys are used (see property `annot.border`). If a number is specified or a dictionary like `{"width": w}`, only the border width will be changed and any dashes will remain unchanged. Conversely, with a dictionary `{"dashes": [...]}`, only line dashing will be changed. To remove dashing and get a contiguous line, specify `{"dashes": []}`.

`setFlags(flags)`

Changes the flags of the annotation. See *Annotation Flags* for possible values and use the `|` operator to combine several.

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

`setColors(d)`

Changes the colors associated with the annotation.

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.

`updateImage()`

Attempts to modify the displayed graphical image such that it coincides with the values currently contained in the `border` and `colors` properties. This is achieved by modifying the contents stream of the associated appearance `XObject`. Not all possible formats of content streams are currently supported: if the stream contains invocations of yet other `XObject` objects, a `ValueError` is raised.

`fileInfo()`

Returns basic information of an attached file (file attachment annotations only).

Return type dict

Returns a dictionary with keys `filename`, `size` (uncompressed file size), `length` (compressed length).

`fileGet()`
Returns the uncompressed content of the attached file.

Return type bytes or str (Py2)

Returns the content of the attached file.

`fileUpd(`*buffer*, *filename=None*`)`
Updates the content of an attached file with new data. Optionally, the filename can be changed, too.

Parameters

- `buffer` (*bytes or bytearray*) – the new file content.
- `filename` (*str*) – new filename to associate with the file.

Return type int

Returns zero

`parent`
The owning page object of the annotation.

Return type *Page*

`rect`
The rectangle containing the annotation in untransformed coordinates.

Return type *Rect*

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

Return type `Annot`

`type`
Meaningful for PDF only: 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. `[]` if not PDF. See the appendix *Annotation Types* for a list of possible values and their meanings.

Return type list

`info`
Meaningful for PDF only: A dictionary containing various information. All fields are unicode or strings (Python 2 or Python 3 respectively).

- `name` - e.g. for `[12, 'Stamp']` type annotations it will contain the stamp text like `Sold` or `Experimental`.
- `content` - a string containing the text for type `Text` and `FreeText` annotations. Commonly used for filling the text field of annotation pop-up windows. For `FileAttachment` it should be used as description for the attached file. Initially just contains the filename.
- `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`
Meaningful for PDF only: An integer whose low order bits contain flags for how the annotation should be presented. See section *Annotation Flags* for details.

Return type int

`lineEnds`
Meaningful for PDF only: A dictionary specifying the starting and the ending appearance of annotations of types `Line`, `PolyLine`, among others. An example would be `{'start': 'None', 'end': 'OpenArrow'}`. `{}` if not specified or 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 dict

`vertices`
Meaningful for PDF only: A list containing point ("vertices") coordinates (each given by 2 floats specifying the x and y coordinate respectively) for various types of annotations:

- `Line` - the starting and ending coordinates (4 floats).
- `[2, 'FreeText', 'FreeTextCallout']` - 4 or 6 floats designating the starting, the (optional) knee point, and the ending coordinates.
- `PolyLine` / `Polygon` - the coordinates of the edges connected by line pieces (`2 * n` floats for `n` points).
- text markup annotations - `8 * n` floats specifying the `QuadPoints` of the `n` marked text spans (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`
Meaningful for PDF only: A dictionary of two lists of floats in range `0 <= float <= 1` specifying the common (`common`) or `stroke` and the interior (`fill`) `non-stroke` colors. The common 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 and color `red`. Both lists can be `[]` if not specified. The dictionary will be empty `{}` if no PDF. The value of each float is mapped to integer values from `0 (<=> 0.0)` to `255 (<=> 1.0)`.

Return type dict

`border`
Meaningful for PDF only: A dictionary containing border characteristics. It will be empty `{}` if not PDF or when no border information is provided. Technically, the PDF entries `/Border`, `/BS` and `/BE` will be checked to build this information. The following keys can occur:

- `width` - a float indicating the border thickness in points.
- `effect` - a list specifying a border line effect like `[1, 'C']`. The first entry "intensity" is an integer (from 0 to 2 for maximum intensity). The second is either 'S' for "no effect" or 'C' for a "cloudy" line.
- `dashes` - a list of integers (arbitrarily limited to 10) specifying a line dash pattern in user units (usually points). `[]` 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.

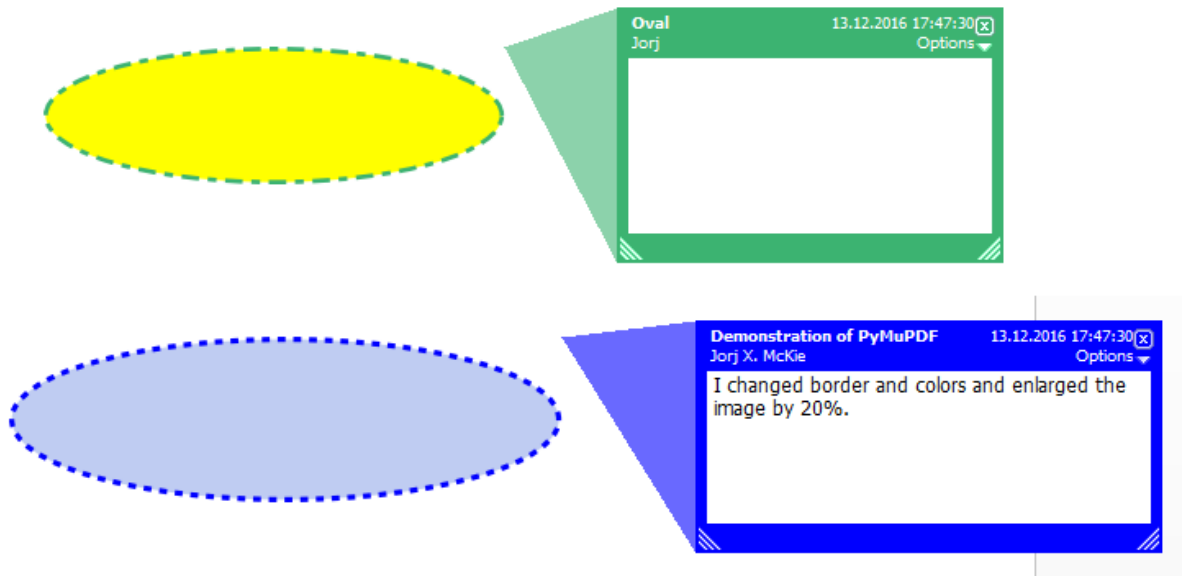
Return type dict

4.1.1 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 border / popup color to blue and fill color to some light blue
annot.setColors({"common":[0, 0, 1], "fill":[0.75, 0.8, 0.95]})
info = annot.info                       # get info dict
info["title"] = "Jorj X. McKie"         # author name in popup title

# text in popup window ...
info["content"] = "I changed border and colors and enlarged the image by 20%."
info["subject"] = "Demonstration of PyMuPDF"     # some readers 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
annot.updateImage()                     # update appearance
doc.save("circle-out.pdf", garbage=4)   # save
```

This is how the circle annotation looks like, before and after the change:



4.2 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)`

4.3 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()	decrypt the document
Document.close()	close the document
Document.copyPage()	PDF only: copy a page to another location
Document.deletePage()	PDF only: delete a page by its number
Document.deletePageRange()	PDF only: delete a range of pages
Document.embeddedFileAdd()	PDF only: add a new embedded file from buffer
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
Document.embeddedFileSetInfo()	PDF only: change metadata of an embedded file
Document.getPageFontList()	make a list of fonts on a page
Document.getPageImageList()	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.getToC()	create a table of contents
Document.insertPage()	PDF only: insert a new page
Document.insertPDF()	PDF only: insert pages from another PDF
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.save()	PDF only: save the document
Document.saveIncr()	PDF only: save the document incrementally
Document.searchPageFor()	search for a string on a page

Continued on next page

Table 4.1 – continued from previous page

Method / Attribute	Short Description
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.embeddedFileCount	number of embedded files
Document.isClosed	has document been closed?
Document.isPDF	is document type PDF?
Document.metadata	metadata
Document.name	filename of document
Document.needsPass	require password to access data?
Document.openErrCode	> 0 if repair occurred during open
Document.openErrMsg	last error message if openErrCode > 0
Document.outline	first *Outline* item
Document.pageCount	number of pages
Document.permissions	permissions to access the document

Class API

`class Document`

`__init__`(*self*[, *filename*])
Constructs a `Document` object from `filename`.

Parameters `filename` (*str*) – A string containing the path / name of the document file to be used. The file will be opened and remain open until either explicitely closed (see below) or until end of program. If omitted or `None`, a new empty **PDF** document will be created.

Return type `Document`

Returns A `Document` object.

`__init__`(*self*, *filetype*, *stream*)
Constructs a `Document` object from memory area `stream`.

Parameters

- `filetype` (*str*) – A string specifying the type of document contained in `stream`. This may be either something that looks 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`. Recommended is using the filename scheme, or even the name of the original file for documentation purposes. But just using strings like `"pdf"` will also work.
- `stream` (*bytes*) – A memory area representing the content of a supported document type. A type of `bytearray` is supported, too.

Return type `Document`

Returns A `Document` object.

`authenticate`(*password*)
Decrypts the document with the string `password`. If successful, all of the document's data can be accessed (e.g. for rendering).

Parameters `password` (*str*) – The password to be used.

Return type int

Returns `True (1)` if decryption with `password` was successful, `False (0)` otherwise. If successfull, indicator `isEncrypted` is set to `False`.

`loadPage(pno = 0)`
Loads a `Page` for further processing like rendering, text searching, etc. See the *Page* object.

Parameters `pno` (*int*) – page number, zero-based (0 is default and the first page of the document) and `< doc.pageCount`. If `pno < 0`, then page `pno % pageCount` will be loaded (IAW `pageCount` will be added to `pno` until the result is no longer negative). 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: Conveniently, pages can also be loaded via indexes over the document: `doc.loadPage(n) == doc[n]`. Consequently, a document can also be used as an iterator over its pages, e.g. `for page in doc: ...` and `for page in reversed(doc): ...` will yield the *Page*s of `doc` as `page`.

`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 (integer). The first entry has hierarchy level 1, and entries in a row increase by at most one level.
- title - title (string)
- page - 1-based page number (integer). Page numbers `< 1` either indicate a target outside this document or no target at all (see next entry).
- dest - included only if `simple = False` is specified. A dictionary containing details of the link destination.

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

Return type *Pixmap*

`getPageImageList(pno)`
PDF only: Return a list of all image descriptions referenced by a page.

Parameters `pno` (*int*) – page number, zero-based. Any value `< len(doc)` is acceptable.

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]`. Where `xref` is the image object number, `smask` is the object number of its soft-mask image (if present), `width` and `height` are the image dimensions, `bpc` denotes the number of bits per component (a typical value is 8), `colorspace` a string naming the colorspace (like `DeviceRGB`), `alt. colorspace` is any alternate colorspace depending on the value of `colorspace`, and `name` - which is the symbolic name *(str)* by which the page references this particular image in its content stream. See below how this information can be used to extract PDF images as separate files. Another demonstration:

```
>>> doc = fitz.open("pymupdf.pdf")
>>> imglist = doc.getPageImageList(0)
>>> for img in imglist: print img
((241, 0, 1043, 457, 8, 'DeviceRGB', '', 'Im1'))
>>> pix = fitz.Pixmap(doc, 241)
>>> pix
fitz.Pixmap(DeviceRGB, fitz.IRect(0, 0, 1043, 457), 0)
```

`getPageFontList(`*pno*`)`

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

Parameters `pno` (*int*) – page number, zero-based, any value `< len(doc)`.

Return type list

Returns a list of fonts referenced by this page. Each entry looks like `(xref, ext, type, basefont, name)`. Where `xref` is the font object number, `ext` font file extension, `type` is the font type (like `Type1` or `TrueType` etc.), `basefont` is the base font name and `name` is the reference name, by which the page references it in its contents stream:

```
>>> doc=fitz.open("pymupdf.pdf")
>>> for f in doc.getPageFontList(85): print(f)
(344, 'pfa', 'Type1', 'HVTNTB+SFSX1000', 'F18')
(343, 'pfa', 'Type1', 'KPGUVC+SFTT1000', 'F16')
(745, 'pfa', 'Type1', 'OBIJJJ+SFRM1440', 'F38')
(470, 'pfa', 'Type1', 'AFLLUK+SFTI1000', 'F49')
(342, 'pfa', 'Type1', 'GWNVMD+SFRM1000', 'F15')
(341, 'pfa', 'Type1', 'MFMRXE+SFBX1000', 'F41')
(523, 'pfa', 'Type1', 'LDRDRB+SFIT1000', 'F74')
```

Note: Fonts are stored on the document level (like images). The reference name is specific for the page. Other pages may use a different name for the same font. Also note, that a font may appear in this list allthough no text actually uses it. But conversely, every piece of text on the page will refer to exactly one of these entries. Look here for the meaning of *Font File Extensions*.

Note: For more background see *Adobe PDF Reference 1.7* chapters 5.4 to 5.8, pp 410.

`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, zero-based. Any value `< len(doc)` is acceptable.
- `output` (*str*) – A string specifying the requested output format: text, html, json or xml. Default is `text`.

Return type str

`select(`*list*`)`

PDF only: Keeps only those pages of the document whose numbers occur in the list. Empty lists or elements outside the range `0 <= page < doc.pageCount` will cause a `ValueError`. For more details see remarks at the bottom or this chapter.

Parameters `list` (*sequence*) – A list (or tuple) of page numbers (zero-based) to be included. Pages not in the list 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 list exactly as specified.

Return type int

Returns Zero upon successful execution. All document information will be updated to reflect the new state of the document, like outlines, number and sequence of pages, etc. Changes become permanent only after saving the document. Incremental save is supported.

`setMetadata(`*m*`)`

PDF only: Sets or updates the metadata of the document as specified in `m`, a Python dictionary. As with method `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, these keys therefore have no effect 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 `m = {}` 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 for this method.

Return type int

Returns Zero upon successful execution and `doc.metadata` will be updated.

`setToC(`*toc*`)`

PDF only: Replaces the **complete current outline** tree (table of contents) with a new one. 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.

Please note, that currently the `is_open` flag is set to `False`. Therefore all entries other than level 1 will initially be shown collapsed in PDF readers.

- 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.
  - `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.

`save(`*outfile*, *garbage=0*, *clean=0*, *deflate=0*, *incremental=0*, *ascii=0*, *expand=0*, *linear=0*`)`

PDF only: Saves the document in its **current state** under the name `outfile`. A document may have changed for a number of reasons: e.g. after a successful `authenticate`, a decrypted copy will be saved, and, in addition (even without optional parameters), some basic cleaning may also have occurred, e.g. broken xref tables may have been repaired and earlier incremental changes may have been resolved. If you executed any modifying methods, their results will also be reflected in the saved version.

Parameters

- `outfile` (*str*) – The file name to save to. Must be different from the original value value if `incremental=False`. When saving incrementally, `garbage` and `linear` **must be** `False / 0` and `outfile` **must equal** the original filename (for convenience use `doc.name`).
- `garbage` (*int*) – Do garbage collection: 0 = none, 1 = remove unused objects, 2 = in addition to 1, compact xref table, 3 = in addition to 2, merge duplicate objects, 4 = in addition to 3, check streams for duplication. Excludes `incremental`.
- `clean` (*int*) – Clean content streams[1]: 0 / False, 1 / True.
- `deflate` (*int*) – Deflate uncompressed streams: 0 / False, 1 / True.
- `incremental` (*int*) – Only save changed objects: 0 / False, 1 / True. Excludes `garbage` and `linear`. Cannot be used for decrypted files and for files opened in repair mode (`openErrCode > 0`). In these cases saving to a new file is required.
- `ascii` (*int*) – Where possible make the output ASCII: 0 / False, 1 / True.
- `expand` (*int*) – Decompress contents: 0 = none, 1 = images, 2 = fonts, 255 = all. This convenience option generates a decompressed file version that can be better read by some other programs.
- `linear` (*int*) – Save a linearised version of the document: 0 = False, 1 = True. This option creates a file format for improved performance when read via internet connections. Excludes `incremental`.

Return type int

Returns Zero upon successful execution.

`saveIncr()`

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

Caution: A PDF may not be encrypted, but still be password protected against changes - see the `permissions` property. Performing incremental saves if `permissions["edit"] == False` can lead to unpredictable results. Save to a new file in such a case. We also consider raising an exception under this condition.

`searchPageFor(`*pno*, *text*, *hit_ max = 16*`)`

Search for `text` on page number `pno`. Works exactly like the corresponding *Page.searchFor()*. Any integer `pno < len(doc)` is acceptable.

[1] 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.

write(*garbage=0*, *clean=0*, *deflate=0*, *ascii=0*, *expand=0*, *linear=0*)

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 *Document.save()*. The tutorial contains an example for using this method as a pre-processor to pdfrw.

Return type bytes

Returns a bytes object containing the complete document data.

insertPDF(*docsrc*, *from_page = -1*, *to_page = -1*, *start_at = -1*, *rotate = -1*, *links = 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). If you do not specify a value (or `-1`), the original will not be changed. Otherwise it must be an integer multiple of 90 (not checked). Rotation is counter-clockwise if `rotate` is positive, else clockwise.
- `links` (*bool*) – Choose whether (internal and external) links should be included with the copy. Default is `True`. An **internal** link is always excluded if its destination is not one of the copied pages.

Return type int

Returns Zero upon successful execution.

Note: If `from_page > to_page`, pages will be copied in reverse order. If `0 <= from_page == to_page`, then one page will be copied.

Note: `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 in the *examples* directory: it can join PDF documents and at the same time piece together respective parts of the tables of contents.

---

insertPage(*to = -1*, *text = None*, *fontsize = 11*, *width = 595*, *height = 842*, *fontname = "Helvetica"*, *fontfile = None*, *color = (0, 0, 0)*)

PDF only: Insert an empty page. Default page dimensions are those of A4 portrait paper format. Optionally, text can also be inserted - provided as a string or asequence.

Parameters

- `to` (*int*) – page number (0-based) in front of which to insert. Valid specifications must be in range `-1 <= pno <= len(doc)`. The default `-1` and `pno = len(doc)` indicate end of document, i.e. after the last page.
- `text` (*str or sequence*) – optional text to put on the page. If given, it will start at 72 points (one inch) below top and 50 points from left. Line breaks

(\n) will be honored, if it is a string. No care will be taken as to whether lines are too wide. However, text output stops when no more lines will fit on the page (discarding any remaining text). If a sequence is specified, its entries must be a of type string. Each entry will be put on one line. Line breaks *within an entry* will be treated as any other white space. If you want to calculate the number of lines fitting on a page beforehand, use this formula: `int((height - 108) / (fontsize * 1.2)`. So, this methods reserves one inch at the top and 1/2 inches at the bottom of the page as free space.

- `fontsize` (*float*) – font size in pixels. Default is 11. If more than one line is provided, a line spacing of `fontsize * 1.2` (fontsize plus 20%) is used.
- `width` (*float*) – width in pixels. Default is 595 (A4 width). Choose 612 for *Letter width*.
- `height` (*float*) – page height in pixels. Default is 842 (A4 height). Choose 792 for *Letter height*.
- `fontname` (*str*) – name of one of the *PDF Base 14 Fonts* (default is "Helvetica") if fontfile is not specified.
- `fontfile` (*str*) – file path of a font existing on the system. If this parameter is specified, specifying `fontname` is **mandatory**. If the font is new to the PDF, it will be embedded. Of the font file, index 0 is used. Be sure to choose a font that supports horizontal, left-to-right spacing.
- `color` (*sequence*) – RGB text color specified as a triple of floats in range 0 to 1. E.g. specify black (default) as `(0, 0, 0)`, red as `(1, 0, 0)`, some gray value as `(0.5, 0.5, 0.5)`, etc.

Return type int

Returns number of text lines put on the page. Use this to check which part of your text did not fit.

Notes:

This method can be used to

1. create a PDF containing only one empty page of a given dimension. The size of such a file is well below 500 bytes and hence close to the theoretical PDF minimum.
2. create a protocol page of which files have been embedded, or separator pages between joined pieces of PDF Documents.
3. convert textfiles to PDF like in the demo script text2pdf.py.
4. For now, the inserted text should restrict itself to one byte character codes.
5. An easy way to create pages with a usual paper format, use a statement like `width, height = fitz.PaperSize("A4-L")`.
6. To simplify color specification, we provide a *Color Database*. This allows you to specify `color = getColor("turquoise")`, without bothering about any more details.

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

PDF only: Convenience method: insert an empty page like `insertPage()` does. Valid parameters have the same meaning. However, no text can be inserted, instead the inserted page object is returned.

Return type *Page*

Returns the page object just inserted.

`deletePage`(*pno*)

PDF only: Delete a page given by its 0-based number in range `0 <= pno < len(doc)`.

Parameters `pno` (*int*) – the page to be deleted.

deletePageRange(*from_page = -1*, *to_page = -1*)
PDF only: Delete a range of pages specified as 0-based numbers. Any negative parameter will first be replaced by `len(doc) - 1`. After that, condition `0 <= from_page <= to_page < len(doc)` must be true. If the parameters are equal, one page will be deleted.

Parameters

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

copyPage(*pno*, *to = -1*)
PDF only: Copy a page within the document.

Parameters

- `pno` (*int*) – the page to be copied. Number must be in range `0 <= pno < len(doc)`.
- `to` (*int*) – the page number in front of which to insert the copy. To insert at end of document (default), specify a negative value.

movePage(*pno*, *to = -1*)
PDF only: Move (copy and then delete original) page to another location.

Parameters

- `pno` (*int*) – the page to be moved. Number must be in range `0 <= pno < len(doc)`.
- `to` (*int*) – the page number in front of which to insert the moved page. To insert at end of document (default), specify a negative value. Must not be in `(pno, pno + 1)`.

embeddedFileInfo(*n*)
PDF only: Retrieve information of an embedded file identified by either its number or by its name.

Parameters `n` (*int or str*) – index or name of entry. Obviously `0 <= n < embeddedFileCount` must be true if `n` is an integer.

Return type dict

Returns

a dictionary with the following keys:

- `name` - (*str*) name under which this entry is stored
- `file` - (*str*) filename associated with the entry
- `desc` - (*str*) description of the entry
- `size` - (*int*) original content size
- `length` - (*int*) compressed content length

embeddedFileSetInfo(*n*, *filename = filename*, *desc = desc*)
PDF only: Change some information of an embedded file given its entry number or name. At least one of `filename` and `desc` must be specified. Response will be zero if successful, else an exception is raised.

Parameters

- `n` (*int or str*) – index or name of entry. Obviously `0 <= n < embeddedFileCount` must be true if `n` is an integer.
- `filename` (*str*) – sets the filename of the entry.
- `desc` (*str*) – sets the description of the entry.

`embeddedFileGet(n)`
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 `n` (*int or str*) – index or name of entry. Obviously `0 <= n < embeddedFileCount` must be true if `n` is an integer.

Return type `bytes` (Python 3), `str` (Python 2)

`embeddedFileDel(name)`
PDF only: Remove an entry from the portfolio. As always, physical deletion of the embedded file content (and file space regain) will occur when the document is saved to a new file with `garbage` option. With an incremental save, the associated object will only be marked deleted.

Note: We do not support entry **numbers** for this function yet. If you need to e.g. delete **all** embedded files, scan through all embedded files by number, and use the returned dictionary's `name` entry to delete each one. This function will delete the first entry with this name it finds. Be wary that for arbitrary PDF files, this may not have been the only one, because PDF itself has no mechanism to prevent duplicate entries ...

Parameters `name` (*str*) – name of entry.

`embeddedFileAdd(stream, name, filename = filename, desc = desc)`
PDF only: Add new content to the document's portfolio.

Parameters

- `stream` (*bytes or bytearray or str (Python 2 only)*) – contents
- `name` (*str*) – new entry identifier, must not already exist in embedded files.
- `filename` (*str*) – optional filename or `None`, documentation only, will be set to `name` if `None` or omitted.
- `desc` (*str*) – optional description or `None`, arbitrary documentation text, will be set to `name` if `None` or omitted.

Return type int

Returns the index given to the new entry. In the current (April 11, 2017) MuPDF version, this is not reliably true (for this reason we have decided to restrict `embeddedFileDel()` to entries identified by name). Use character string look up to find your entry again. For any error condition, an exception is raised.

`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 / 0` if document is still open, `True / 1` otherwise. 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

`needsPass`
Contains an indicator showing whether the document is encrypted (`True (1)`) or not (`False (0)`). This indicator remains unchanged - even after the document has been authenticated. Precludes incremental saves if set.

Type bool

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

Type bool

`permissions`
Shows the permissions to access the document. Contains a dictionary likes this:

```
>>> doc.permissions
{'print': True, 'edit': True, 'note': True, 'copy': True}
```

The keys have the obvious meaning of permissions to print, change, annotate and copy the document, respectively.

Type dict

`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`, 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 PDF version (e.g. 'PDF-1.6').
- `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 dictionary `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

`openErrCode`
If `openErrCode > 0`, errors have occurred while opening / parsing the document, which usually means document structure issues. In this case incremental save cannot be used.

Type int

`openErrMsg`
Contains either an empty string or the last open error message if `openErrCode > 0`. Together with any other error messages of MuPDF's C library, it will also appear on `SYSERR`.

Type str

`embeddedFileCount`
Contains the number of files in the embedded / portfolio files list (also known as collection or attached files). If the document is not a PDF, `-1` will be returned.

Type int

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 and annotations), variables holding old page counts, tables of content and the like. Remember to keep such variables up to date or delete orphaned objects.

4.3.1 Remarks on `select()`

Page numbers in the list need not be unique nor be in any particular sequence. This makes the method a versatile utility to e.g. select only the even or the odd pages, re-arrange a document from back to front, duplicate it, and so forth. In combination with text search or extraction you can also omit / include pages with no text or containing a certain text, etc.

You can execute several selections in a row. The document structure will be updated after each method execution.

Any of those changes will become permanent only with a `doc.save()`. If you have de-selected many pages, consider specifying the `garbage` option to eventually reduce the resulting document's size (when saving to a new file).

Also note, that this method **preserves all links, annotations and bookmarks** that are still valid. In other words: deleting pages only deletes references which point to de-selected pages. Page number of bookmarks (outline items) are automatically updated when a TOC is retrieved again with `getToC()`. If a bookmark's destination page happened to be deleted, then its page number in `getToC()` will be set to `-1`.

The results of this method can of course also be achieved using combinations of methods `copyPage()`, `deletePage()` and `movePage()`. While there are many cases, when these methods are more practical, `select()` is easier and safer to use when many pages are involved.

4.3.2 `select()` Examples

In general, any list of integers within the document's page range can be used. Here are some illustrations.

Delete pages with no text:

```
import fitz
doc = fitz.open("any.pdf")
r = list(range(len(doc)))                    # list of page numbers

for page in doc:
```

```
    if not page.getText():                  # page contains no text
        r.remove(page.number)               # remove page number from list

if len(r) < len(doc):                       # did we actually delete anything?
    doc.select(r)                           # apply the list
doc.save("out.pdf", garbage = 4)            # save result to new PDF, OR

# update the original document ... *** VERY FAST! ***
doc.saveIncr()
```

Create a sub document with only the odd pages:

```
>>> import fitz
>>> doc = fitz.open("any.pdf")
>>> r = list(range(0, len(doc), 2))
>>> doc.select(r)                                # apply the list
>>> doc.save("oddpages.pdf", garbage = 4)        # save sub-PDF of the odd pages
```

Concatenate a document with itself:

```
>>> import fitz
>>> doc = fitz.open("any.pdf")
>>> r = list(range(len(doc)))
>>> r += r                                       # turn PDF into a copy of itself
>>> doc.select(r)
>>> doc.save("any+any.pdf")                      # contains doubled <any.pdf>
```

Create document copy in reverse page order (well, don't try with a million pages):

```
>>> import fitz
>>> doc = fitz.open("any.pdf")
>>> r = list(range(len(doc) - 1, -1, -1))
>>> doc.select(r)
>>> doc.save("back-to-front.pdf")
```

4.3.3 `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
0
>>> 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
0
>>> doc.save("anonymous.pdf", garbage = 4)        # save anonymized doc
0
```

4.3.4 setToC() Example

This shows how to modify or add a table of contents. Also have a look at csv2toc.py and toc2csv.py 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]
```

4.3.5 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(simple = False)      # save TOC 1
>>> toc2 = doc2.getToC(simple = 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 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)
```

4.3.6 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)
```

4.4 Identity

Identity is just a *Matrix* that performs no action, to be used whenever the syntax requires a *Matrix*, but no actual transformation should take place.

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 do-nothing matrix as a starting point, use `fitz.Matrix(1, 1)` or `fitz.Matrix(0)` instead, like so:

```
>>> fitz.Matrix(0).preTranslate(2, 5)
fitz.Matrix(1.0, 0.0, -0.0, 1.0, 2.0, 5.0)
```

4.5 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 `IRects`.

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.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.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*, *list*)

Overloaded constructors. Also see examples below and those for the *Rect* class.

If another `irect` is specified, a **new copy** will be made.

If `list` is specified, it must be a Python sequence type of 4 integers. 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 one of

the rectangles is infinite, the other one is taken as the result - and hence also infinite if both rectangles were infinite.

Parameters `ir` (*IRect*) – Second rectangle.

`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. 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 `r` (`IRect` or *Rect*) have a non-empty rectangle in common. This will always be `False` if either is infinite or empty.

Parameters `r` (*IRect* or *Rect*) – the rectangle to check.

Return type bool

`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*

`width`
Contains the width of the bounding box. Equals `x1 - x0`.

Type int

`height`
Contains the height of the bounding box. Equals `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

4.5.1 Remark

A rectangle's coordinates can also be accessed via index, e.g. `r.x0 == r[0]`, and the `tuple()` and `list()` functions yield sequence objects of its components.

4.5.2 IRect Algebra

Algebra provides handy ways to perform inclusion and intersection checks between Rects, IRects and Points. For a general background, see chapter *Operator Algebra for Geometry Objects*.

4.5.3 Examples

Example 1:

```
>>> ir = fitz.IRect(10, 10, 410, 610)
>>> ir
fitz.IRect(10, 10, 410, 610)
>>> ir.height
600
>>> ir.width
400
>>> ir.getArea('mm')      # calculate area in square millimeters
29868.51852
```

Example 2:

```
>>> m = fitz.Matrix(45)
>>> ir = fitz.IRect(10, 10, 410, 610)
>>> ir * m                          # rotate rectangle by 45 degrees
fitz.IRect(-425, 14, 283, 722)
>>>
>>> ir | fitz.Point(5, 5)           # enlarge rectangle to contain a point
fitz.IRect(5, 5, 410, 610)
>>>
>>> ir + 5                          # shift the rect by 5 points
fitz.IRect(15, 15, 415, 615)
>>>
```

```
>>> ir & fitz.Rect(0.0, 0.0, 15.0, 15.0)
fitz.IRect(10, 10, 15, 15)
>>> ir /= (1, 2, 3, 4, 5, 6)        # divide by a matrix
>>> ir
fitz.IRect(-14, 0, 4, 8)
```

Example 3:

```
>>> # test whether two rectangle are disjoint
>>> if not r1.intersects(r2): print("disjoint rectangles")
>>>
>>> # test whether r2 containes x (x is point-like or rect-like)
>>> if r2.contains(x): print("x is contained in r2")
>>>
>>> # or even simpler:
>>> if x in r2: print("x is contained in r2")
```

4.6 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.rect	clickable area in untransformed coordinates.
Link.uri	link destination
Link.isExternal	external link destination?
Link.next	points to next link
Link.dest	points to link destination details

Class API

`class Link`

`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

`next`
The next `Link` or `None`

Type `Link`

`dest`
The link destination details object.

Type *linkDest*

4.7 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

4.8 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.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

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*, *list*)

Overloaded constructors.

Without parameters, `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), respectively.

`matrix` specifies another `Matrix` from which a **new copy** will be made.

Float value `degree` specifies the creation of a rotation matrix.

Python sequence `list` (list, tuple, etc.) must contain exactly 6 values when specified. Non-numeric entries will raise an exception.

`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]`.

`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*`)`
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.

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

4.8.1 Remarks 1

For a matrix `m`, properties `a` to `f` can also be accessed by index, e.g. `m.a == m[0]` and `m[0] = 1` has the same effect as `m.a = 1`. The `tuple()` and `list()` functions yield sequence objects of its components.

Language constructs like `x in m` is equal to `x in tuple(m)`.

4.8.2 Remarks 2

Changes of matrix properties and execution of matrix methods can be executed consecutively. This is the same as multiplying the respective matrices.

Matrix multiplications are **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 transformations, Adobe recommends the following approach (*Adobe PDF Reference 1.7*, page 206):

1. Shift ("translate")
2. Rotate
3. Scale or shear ("skew")

4.8.3 Matrix Algebra

For a general background, see chapter *Operator Algebra for Geometry Objects*.

This makes the following operations possible:

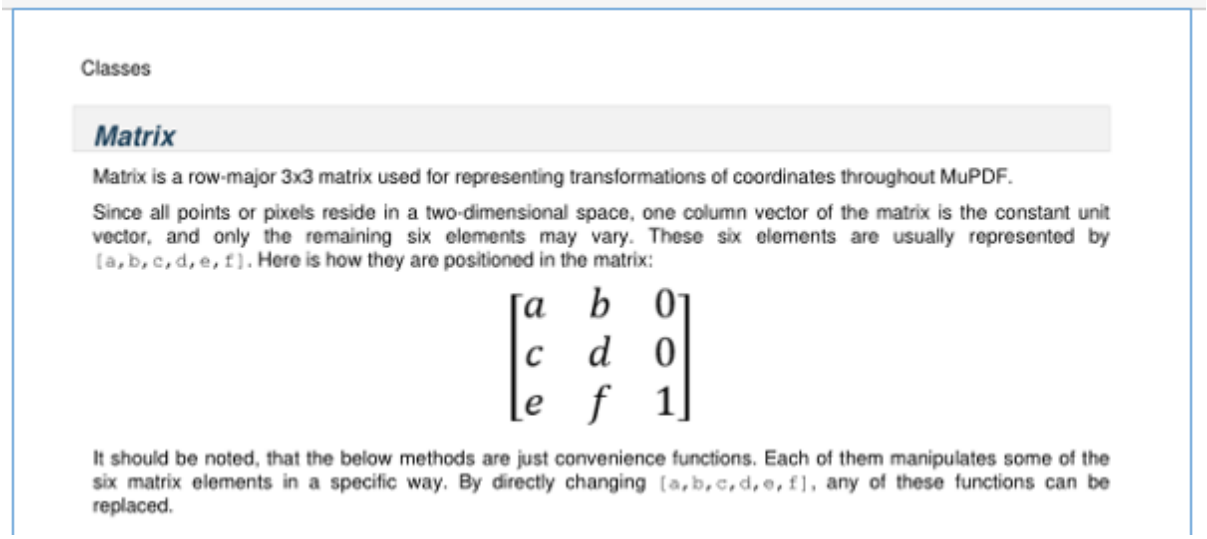
```
>>> m45p = fitz.Matrix(45)            # rotate 45 degrees clockwise
>>> m45m = fitz.Matrix(-45)           # rotate 45 degrees counterclockwise
>>> m90p = fitz.Matrix(90)            # rotate 90 degrees clockwise
>>>
>>> abs(m45p * ~m45p - fitz.Identity) # should be (close to) zero:
8.429369702178807e-08
>>>
>>> abs(m90p - m45p * m45p)           # should be (close to) zero:
8.429369702178807e-08
>>>
>>> abs(m45p * m45m - fitz.Identity)  # should be (close to) zero:
2.1073424255447017e-07
>>>
>>> abs(m45p - ~m45m)                 # should be (close to) zero:
2.384185791015625e-07
```

```
>>>
>>> m90p * m90p * m90p * m90p          # should be 360 degrees = fitz.Identity
fitz.Matrix(1.0, -0.0, 0.0, 1.0, 0.0, 0.0)
```

4.8.4 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:


Classes

Matrix

Matrix is a row-major 3x3 matrix used for representing transformations of coordinates throughout MuPDF.

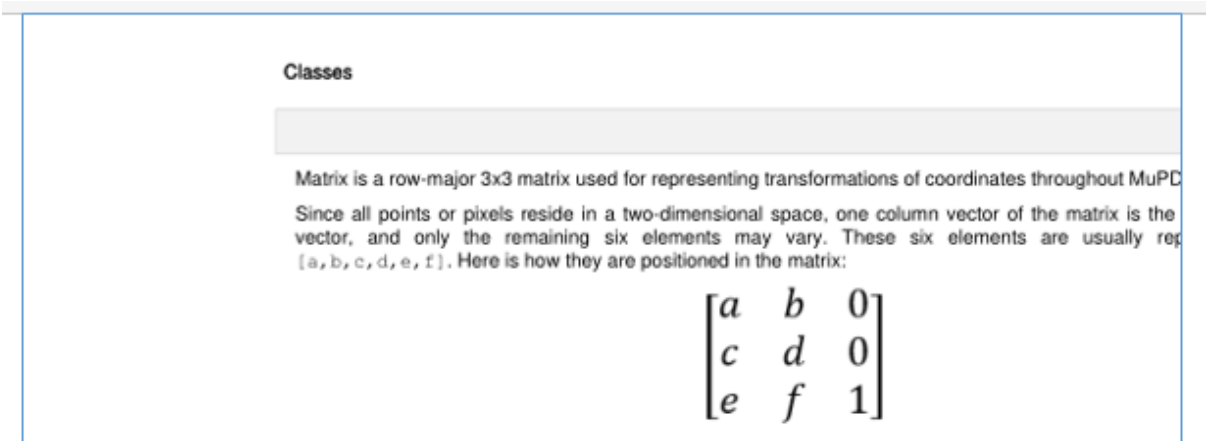
Since all points or pixels reside in a two-dimensional space, one column vector of the matrix is the constant unit vector, and only the remaining six elements may vary. These six elements are usually represented by `[a,b,c,d,e,f]`. Here is how they are positioned in the matrix:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

It should be noted, that the below methods are just convenience functions. Each of them manipulates some of the six matrix elements in a specific way. By directly changing `[a,b,c,d,e,f]`, any of these functions can be replaced.

4.8.5 Shifting

We transform it with a matrix where `e = 100` (right shift by 100 pixels).

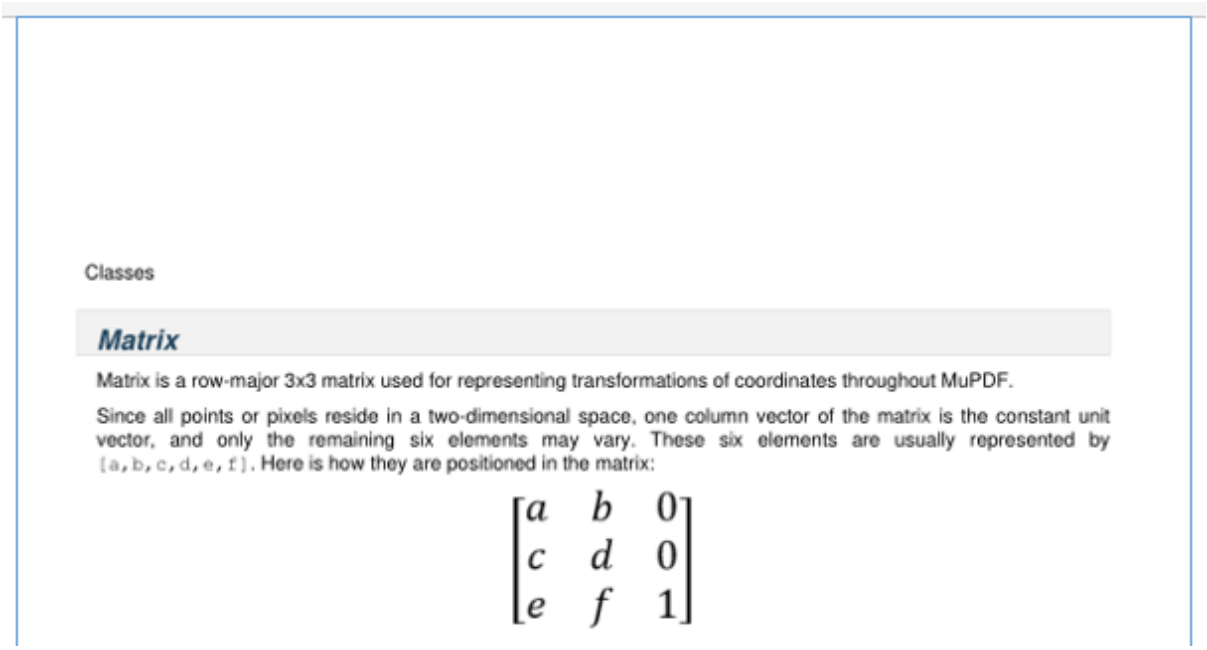

Classes

Matrix is a row-major 3x3 matrix used for representing transformations of coordinates throughout MuPD

Since all points or pixels reside in a two-dimensional space, one column vector of the matrix is the vector, and only the remaining six elements may vary. These six elements are usually rep `[a,b,c,d,e,f]`. Here is how they are positioned in the matrix:

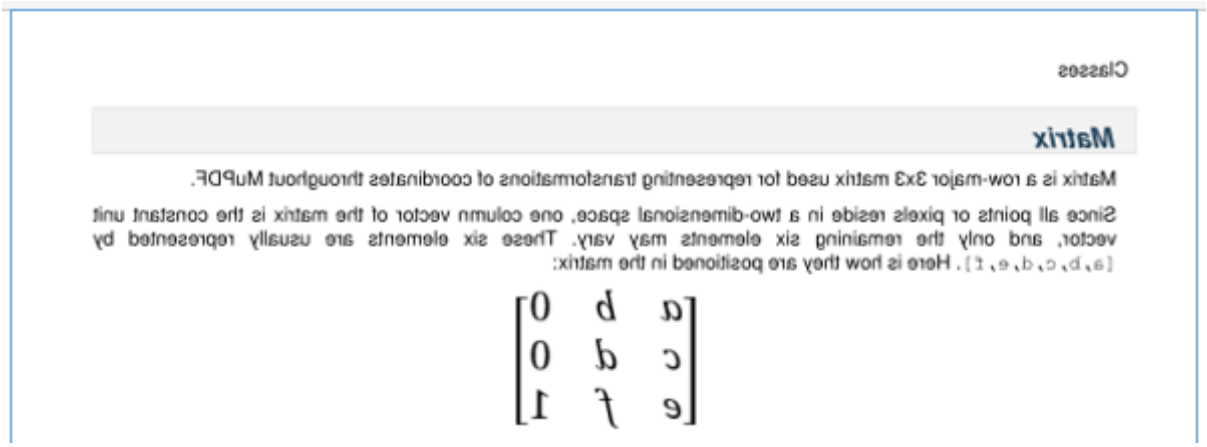
$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

Next we do a down shift by 100 pixels: `f = 100`.

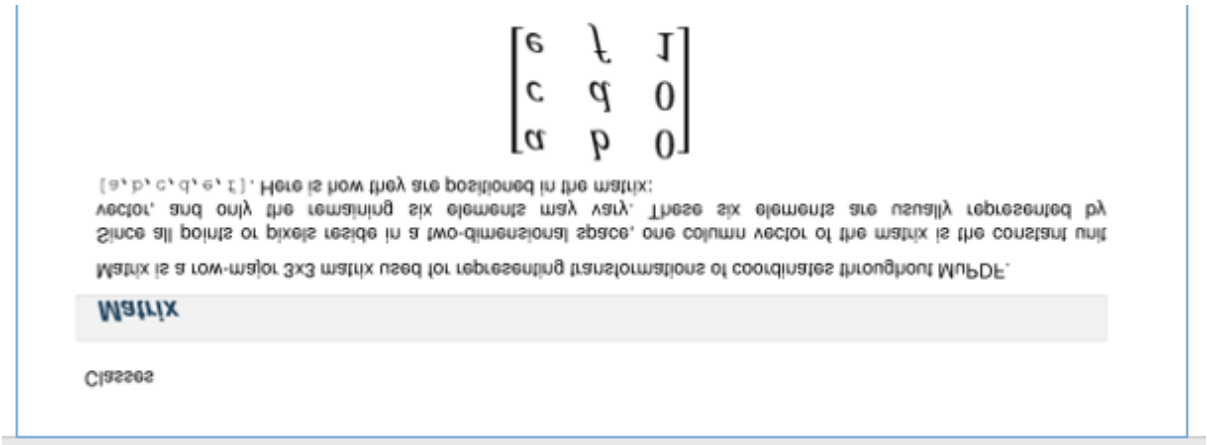


4.8.6 Flipping

Flip the page left-right (`a = -1`).

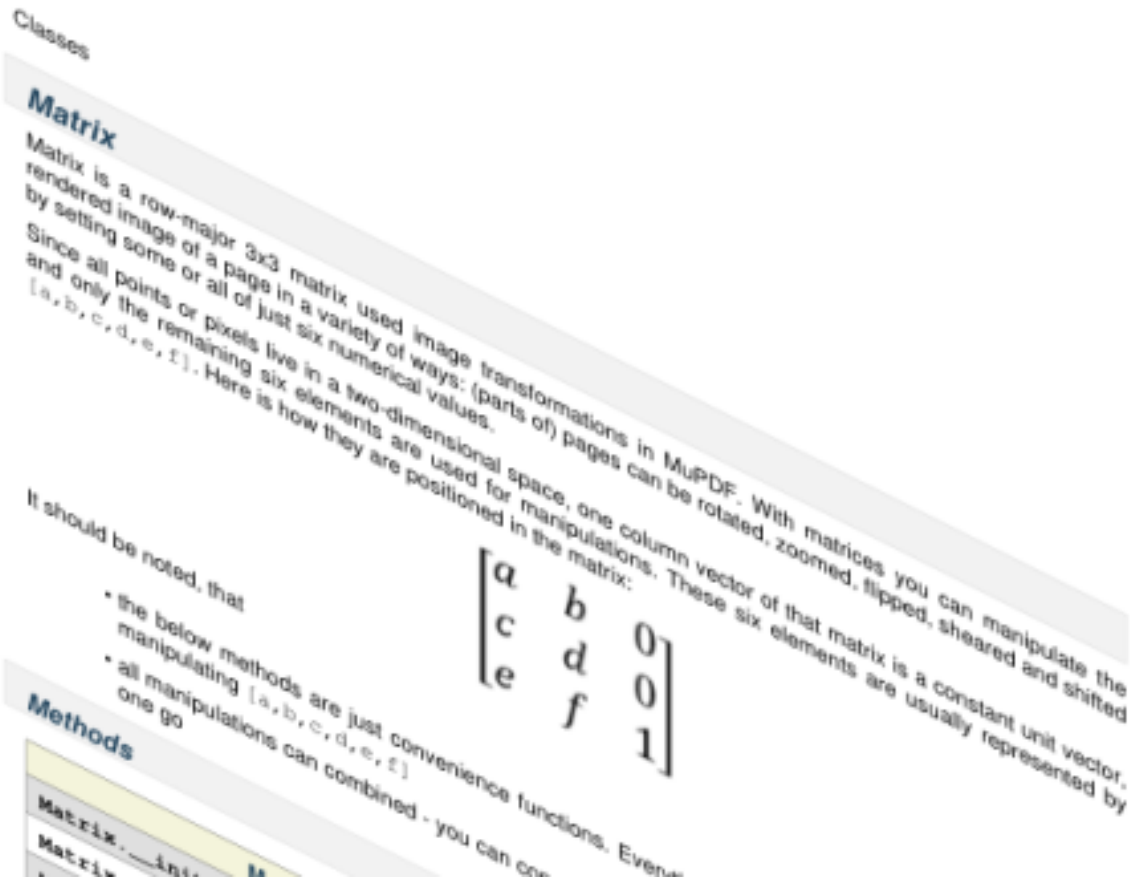


Flip up-down (`d = -1`).

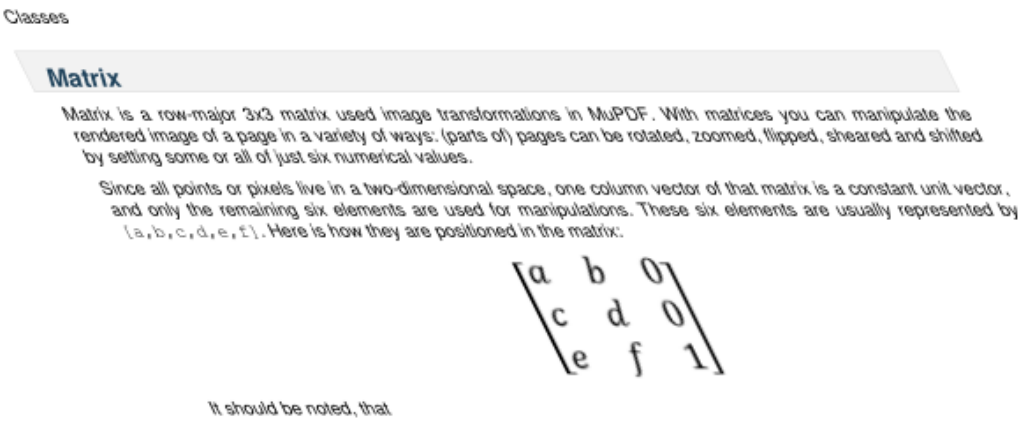


4.8.7 Shearing

First a shear in Y direction (`b = 0.5`).

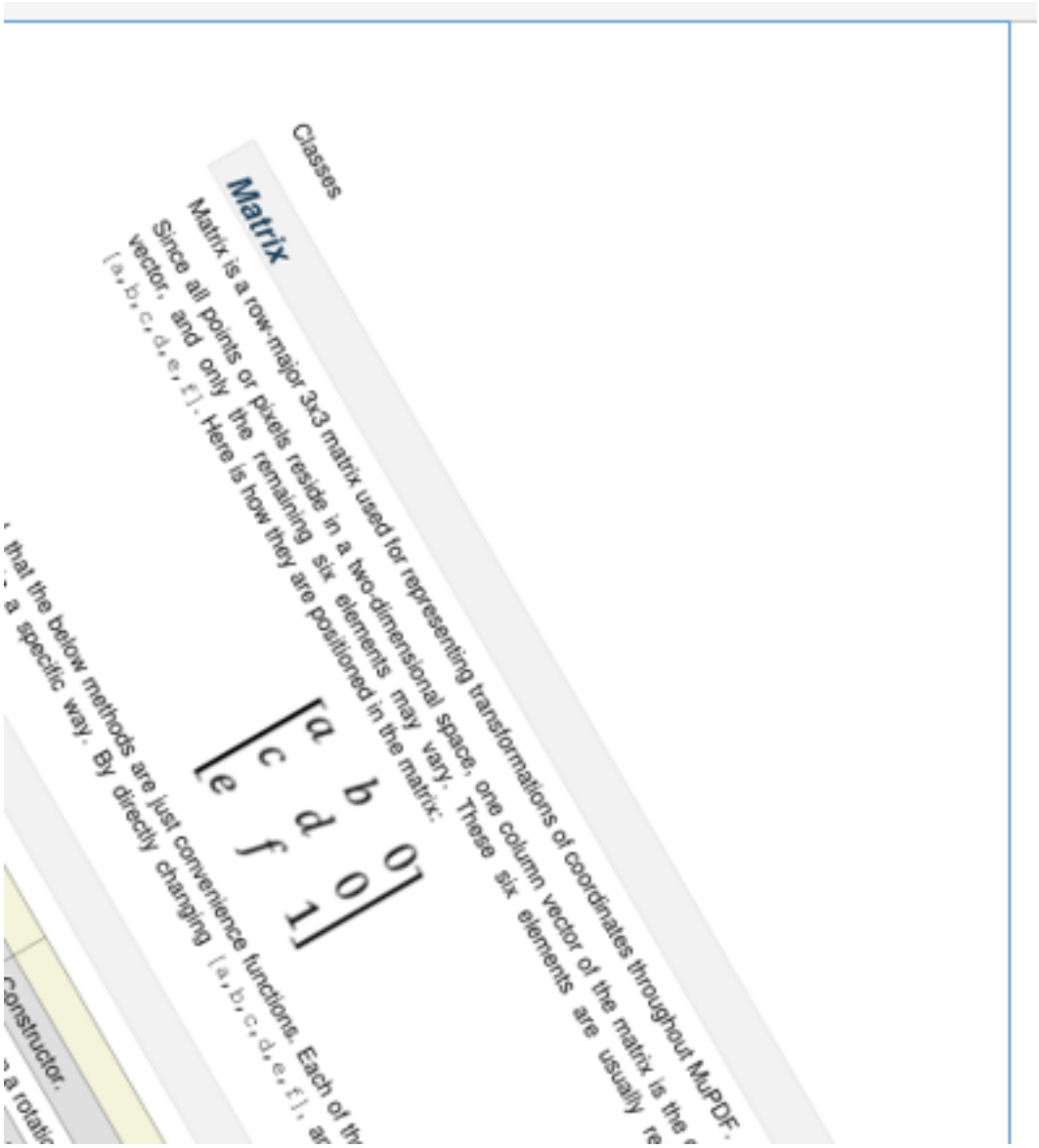


Second a shear in X direction (`c = 0.5`).



4.8.8 Rotating

Finally a rotation by 30 clockwise degrees (`preRotate(-30)`).



4.9 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*

4.10 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. If a page property or method is being used, an exception is raised saying that the page object is "orphaned".

Several page methods have a *Document* counterpart for convenience. At the end of this chapter you will find a synopsis.

Methods `insertText()`, `insertTextbox()` and `draw*()` are for PDF pages only. They provide "stand-alone" shortcut versions for the same-named methods of the *Shape* class. For detailed descriptions have a look in that chapter.

- 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()*.

Method / Attribute	Short Description
Page.bound()	rectangle (mediabox) of the page
Page.deleteAnnot()	PDF only: delete an annotation
Page.deleteLink()	PDF only: delete a link
Page.drawBezier()	PDF only: draw a cubic Bézier curve
Page.drawCircle()	PDF only: draw a circle
Page.drawCurve()	PDF only: draw a special Bézier 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.getImageList()	PDF only: get list of used images
Page.getLinks()	get all links
Page.getPixmap()	create a *Pixmap*
Page.getSVGimage()	convert page image to SVG format
Page.getText()	extract the page's text
Page.getTextBlocks()	extract text blocks as a Python list
Page.getTextWords()	extract text words as a Python list
Page.insertImage()	PDF only: insert an image
Page.insertLink()	PDF only: insert a new link
Page.insertText()	PDF only: insert text
Page.insertTextbox()	PDF only: insert a text box
Page.loadLinks()	return the first link on a page
Page.newShape()	PDF only: start a new *Shape*
Page.searchFor()	search for a string
Page.setRotation()	PDF only: set page rotation
Page.showPDFpage()	PDF only: display PDF page image
Page.updateLink()	PDF only: modify a link
Page.CropBoxPosition	top-left point of /CropBox
Page.MediaBoxSize	bottom-right point of /MediaBox
Page.firstAnnot	first *Annot* on the page
Page.firstLink	first *Link* on the page
Page.number	page number
Page.parent	owning document object
Page.rect	rectangle (mediabox) of the page
Page.rotation	PDF only: page rotation

Class API

`class Page`

`bound()`
Determine the rectangle ("mediabox", before transformation) of the page.

Return type *Rect*

`deleteAnnot(annot)`
PDF only: Delete the specified annotation from the page and (for all document types) return the next one.

Parameters `annot` (*Annot*) – the annotation to be deleted.

Return type *Annot*

Returns the next annotation of the deleted one.

`deleteLink(linkdict)`
PDF only: Delete the specified link from the page. The parameter must be a dictionary of format as provided by the `getLinks()` method (see below).

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 the `getLinks()` method (see below).

Parameters `linkdict` (*dict*) – the link to be inserted.

`updateLink(linkdict)`
PDF only: Modify the specified link. The parameter must be a dictionary of format as provided by the `getLinks()` method (see below).

Parameters `linkdict` (*dict*) – the link to be modified.

`getLinks()`
Retrieves **all** links of a page.

Return type list

Returns A list of dictionaries. The entries are in the order as specified during PDF generation. For a description of the dictionary entries see below. Always use this method if you intend to make changes to the links of a page.

`insertText(point, text = text, fontsize = 11, fontname = "Helvetica", fontfile = None, idx = 0, color = (0, 0, 0), rotate = 0, morph = None, overlay = True)`
PDF only: Insert text.

`insertTextbox(rect, buffer, fontsize = 11, fontname = "Helvetica", fontfile = None, idx = 0, color = (0, 0, 0), expandtabs = 8, align = TEXT_ALIGN_LEFT, charwidths = None, rotate = 0, morph = None, overlay = True)`
PDF only: Insert text into the specified rectangle.

`drawLine(p1, p2, color = (0, 0, 0), width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`
PDF only: Draw a line from *Point* objects `p1` to `p2`.

`drawZigzag(p1, p2, breadth = 2, color = (0, 0, 0), width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`
PDF only: Draw a zigzag line from *Point* objects `p1` to `p2`.

`drawSquiggle(p1, p2, breadth = 2, color = (0, 0, 0), width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`
PDF only: Draw a squiggly (wavy, undulated) line from *Point* objects `p1` to `p2`.

`drawCircle(center, radius, color = (0, 0, 0), fill = None, width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`
PDF only: Draw a circle around `center` with a radius of `radius`.

`drawOval(rect, color = (0, 0, 0), fill = None, width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`
PDF only: Draw an oval (ellipse) within the given rectangle.

`drawSector`(*center*, *point*, *angle*, *color = (0, 0, 0)*, *fill = None*, *width = 1*, *dashes = None*, *roundCap = True*, *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).

`drawPolyline`(*points*, *color = (0, 0, 0)*, *fill = None*, *width = 1*, *dashes = None*, *roundCap = True*, *overlay = True*, *closePath = False*, *morph = None*)
PDF only: Draw several connected lines defined by a sequence of points.

`drawBezier`(*p1*, *p2*, *p3*, *p4*, *color = (0, 0, 0)*, *fill = None*, *width = 1*, *dashes = None*, *roundCap = True*, *overlay = True*, *closePath = False*, *morph = None*)
PDF only: Draw a cubic Bézier curve from `p1` to `p4` with the control points `p2` and `p3`.

`drawCurve`(*p1*, *p2*, *p3*, *color = (0, 0, 0)*, *fill = None*, *width = 1*, *dashes = None*, *roundCap = True*, *overlay = True*, *closePath = False*, *morph = None*)
PDF only: This is a special case of `drawBezier()`.

`drawRect`(*rect*, *color = (0, 0, 0)*, *fill = None*, *width = 1*, *dashes = None*, *roundCap = True*, *overlay = True*, *morph = None*)
PDF only: Draw a rectangle.

Note: An efficient way to background-color a PDF page with the old Python paper color is `page.drawRect(page.rect, color = py_color, fill = py_color, overlay = False)`, where `py_color = getColor("py_color")`.

`insertImage`(*rect*, *filename = None*, *pixmap = None*, *overlay = True*)
PDF only: Fill the given rectangle with an image. Width and height need not have the same proportions as the image: it will be adjusted to fit. The image is either taken from a pixmap or from a file - **exactly one** of these parameters **must** be specified.

Parameters

- `rect` (*Rect*) – where to put the image on the page. `rect` must be finite, not empty and be completely contained in the page's rectangle.
- `filename` (*str*) – name of an image file (all MuPDF supported formats - see *Pixmap* chapter).
- `pixmap` (*Pixmap*) – pixmap containing the image. When inserting the same image multiple times, this should be the preferred option, because the overhead of opening the image and decompressing its content will occur every time with the filename option.

For a description of the other parameters see *Common Parameters*.

Returns zero

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
>>> pix = fitz.Pixmap("some.jpg")     # an image file
>>> for page in doc:
        page.insertImage(rect, pixmap = pix)
>>> doc.save(...)
```

Notes:

1. If that same image had already been present in the PDF, then only a reference 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 code against its entries. Generating this MD5 table, however, is done only when triggered by the first image insertion - which therefore may have an extended response time.

2. You can use this method to provide a background image for the page, like a copyright, a watermark or a background color. Or you can combine it with `searchFor()` to achieve a textmarker effect.
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 content is stored in its original size - which may be much bigger than the size you want to get displayed. Consider decreasing the stored image size by using the pixmap option and then shrinking it or scaling it down (see *Pixmap* chapter). The file size savings can be very significant.

`getText(`*output = 'text'*`)`

Retrieves the text of a page. Depending on the output parameter, the results of the *TextPage* extract methods are returned.

If `'text'` is specified, plain text is returned **in the order as specified during PDF creation** (which is not necessarily the normal reading order). This may not always look as expected, consider using (and probably modifying) the example program PDF2TextJS.py. It tries to re-arrange text according to the Western reading layout convention "from top-left to bottom-right".

Parameters `output` (*str*) – A string indicating the requested text format, one of `"text"` (default), `"html"`, `"json"`, `"xml"` or `"xhtml"`.

Return type string

Returns The page's text as one string.

Note: Use this method to convert the document into a valid HTML version by wrapping it with appropriate header and trailer strings, see the following snippet. Creating XML, XHTML or JSON documents works in exactly the same way. For XML and JSON you may also include an arbitrary filename like so: `fitz.ConversionHeader("xml", filename = doc.name)`. Also see *Controlling Quality of HTML Output*.

```
>>> doc = fitz.open(...)
>>> ofile = open(doc.name + ".html", "w")
>>> ofile.write(fitz.ConversionHeader("html"))
>>> for page in doc: ofile.write(page.getText("html"))
>>> ofile.write(fitz.ConversionTrailer("html"))
>>> ofile.close()
```

`getTextBlocks(`*images = False*`)`

Extract all text blocks as a Python list. Provides basic positioning information without the need to interpret the output of *TextPage.extractJSON()* or *TextPage.extractXML()*. The block sequence is as specified in the document. All lines of a block are concatenated into one string, separated by a space.

Parameters `images` (*bool*) – also extract image blocks. Default is false. This serves as a means to get complete page layout information. Only metadata, not the image data itself is extracted. Use *TextPage.extractJSON()* for accessing this information.

Return type list

Returns

a list whose items have the following entries.

- `x0, y0, x1, y1`: 4 floats defining the bbox of the block.
- `text`: concatenated text lines in the block *(str)*. If this is an image block, a text like this is contained: `<image: DeviceRGB, width 511, height 379, bpc 8>` (original image's width and height).

- `block_n`: 0-based block number *(int)*.
- `type`: block type *(int)*, 0 = text, 1 = image.

`getTextWords()`
Extract all words as a Python list. Provides positioning information for words without having to interpret the output of *TextPage.extractXML()*. The word sequence is as specified in the document. The accompanying rectangle coordinates can be used to re-arrange the final text output to your liking. Block and line numbers help keeping track of the original position.

Return type list

Returns

a list whose items are lists with the following entries:

- `x0, y0, x1, y1`: 4 floats defining the bbox of the word.
- `word`: the word, spaces stripped off *(str)*. Note that any non-space character is accepted as part of a word - not just letters. So, `Hello world!` will yield the two words `Hello` and `world!`.
- `block_n, line_n, word_n`: 0-based numbers for block, line and word *(int)*.

`getFontList()`
PDF only: Return a list of fonts referenced by the page. Same as *Document.getPageFontList()*.

`getImageList()`
PDF only: Return a list of images referenced by the page. Same as *Document.getPageImageList()*.

`getSVGimage(`*matrix = fitz.Identity*`)`

Create an SVG image from the page. Only full page images are currently supported.

Parameters `matrix` (*Matrix*) – a *Matrix*, default is *Identity*. Valid operations include scaling and rotation.

Returns a UTF-8 encoded string that contains the image. This is XML syntax and can be saved in a text file with extension `.svg`.

`getPixmap(`*matrix = fitz.Identity*, *colorspace = fitz.csRGB*, *clip = None*, *alpha = True*`)`
Create a pixmap from the page. This is probably the most often used method to create pixmaps.

Parameters

- `matrix` (*Matrix*) – a *Matrix*, default is *Identity*.
- `colorspace` (string, *Colorspace*) – Defines the required colorspace, one of `GRAY`, `RGB` or `CMYK` (case insensitive). Or specify a *Colorspace*, e.g. one of the predefined ones: *csGRAY*, *csRGB* or *csCMYK*.
- `clip` (*IRect*) – restrict rendering to the rectangle's area. The default will render the full page.
- `alpha` (*bool*) – A bool indicating whether an alpha channel should be included in the pixmap. Choose `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 in most cases. Also note an important difference in how the image will appear:
  - `True`: pixmap's samples will be pre-cleared with `0x00`, including the alpha byte. This will result in **transparent** areas where the page is empty.

– `False`: pixmap's samples will be pre-cleared with `0xff`. This will result in **white** where the page has nothing to show.

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(`*rot*`)`
PDF only: Sets the rotation of the page.

Parameters `rot` (*int*) – An integer specifying the required rotation in degrees. Should be a (positive or negative) multiple of 90.

Returns zero if successfull, `-1` if not a PDF.

`showPDFpage(`*rect*, *docsrc* = *None*, *pno* = *0*, *keep_proportion* = *True*, *overlay* = *True*, *reuse_xref* = *0*, *clip* = *None*`)`
PDF only: Display the page of another PDF as a **vector image**.

Parameters

- `rect` (*Rect*) – where to place the image.
- `docsrc` (*Document*) – source PDF document containing the page. Must be a different document object, but may be the same file. Either this parameter or a positive value for `reuse_xref` must be specified.
- `pno` (*int*) – page number (0-based) to be displayed.
- `keep_proportion` (*bool*) – control whether to scale width and height synchronously (default).
- `overlay` (*bool*) – put image in foreground (default) or background.
- `reuse_xref` (*int*) – specify an xref number if an already stored page image should be reused. This suppresses copying the source page once more. This

argument takes precedence: if a positive value is given, parameters `docsrc` and `pno` are ignored. A value less than 1 will cause copying page number `pno`.

- `clip` (*Rect*) – choose which part of the input page to show. Default is the complete page.

Returns xref number of the stored page image if successful. Use this result as the value of argument `reuse_xref` when the page should be displayed somewhere else.

Note: This is a multi-purpose method. For instance, it can be used to create "2-up" / "4-up" or posterized versions of existing PDF files (see examples 4-up.py and posterize.py). Or use it to include PDF-based vector images (company logos, watermarks, etc.).

Note: Unfortunately, garbage collection currently does not detect multiple copies of a displayed source page. Therefore, use the `reuse_xref` argument to prevent their creation as follows.

```
>>> # the first showPDFpage will copy the page, subsequent
>>> # executions will refer to the "xref"-ed object.
>>> xref = 0
>>> for page in doc:
        xref = page.showPDFpage(rect, docsrc, pno, reuse_xref = xref)
>>>
```

`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*`)`
Searches for `text` on a page. Identical to *`TextPage.search()`*.

Parameters

- `text` (*str*) – Text to searched for. Upper / lower case is ignored.
- `hit_max` (*int*) – Maximum number of occurrences accepted.

Return type list

Returns A list of *Rect* rectangles each of which surrounds one occurrence of `text`.

`rotation`
PDF only: contains the rotation of the page in degrees and `-1` for other document types.

Type int

`CropBoxPosition`
Contains the top-left coordinates of the page's `/CropBox` for a PDF, otherwise the top-left coordinates of the page's rectangle.

Type *Point*

`MediaBoxSize`

Contains the width and height of the page's `/MediaBox` for a PDF, otherwise the bottom-right coordinates of the page's rectangle.

type *Point*

Note: For non-PDF documents (and in most cases for PDF documents, too) `page.rect == fitz.Rect(page.CropBoxPosition, page.MediaBoxSize)` is true. For PDF documents however, `page.rect` may be a true subset of the `/MediaBox`. In this case these attributes may help to correctly position / evaluate elements of the page.

`firstLink`
Contains the first *Link* of a page (or `None`).

Type *Link*

`firstAnnot`
Contains the first *Annot* of a page (or `None`).

Type *Annot*

`number`
The page number.

Type int

`parent`
The owning document object.

Type *Document*

`rect`
Contains the rectangle ("mediabox", before transformation) of the page. Same as result of method `bound()`.

Type *Rect*

4.10.1 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 cross reference entry 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.

4.10.2 Notes on Supporting Links

MuPDF's support for links has changed in **v1.10a**. These changes affect link types *LINK_GOTO* and *LINK_GOTOR*.

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.

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**).

> **Caution:** 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**.

4.10.3 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(...)
Document._getPageXref(pno)	Page._getXref()

The page number `pno` is 0-based and can be any negative or positive number `< len(doc)`. The document methods invoke their page counterparts via `Document[pno].<method>`.

4.11 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") plus an (optional since v1.10.0) alpha byte.

In PyMuPDF, there exist several ways to create a pixmap. Except one, all of them are available as overloaded constructors. A pixmap can be created ...

1. from a document page (via methods *Page.getPixmap()* or *Document.getPagePixmap()*)
2. empty based on *Colorspace* and *IRect* information
3. from an image file
4. from an in-memory image (bytearray)
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 Types*.

Have a look at the **example** 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()	applie a gamma factor to the pixmap
Pixmap.getPNGData()	return a PNG as a memory area
Pixmap.invertIRect()	invert the pixels of a given area
Pixmap.setAlpha()	sets alpha values
Pixmap.shrink()	reduce size keeping proportions
Pixmap.tintWith()	tint a pixmap with a color
Pixmap.writeImage()	save a pixmap in various 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*)

Empty pixmap: Create an empty pixmap of size and origin given by a rectangle. So, for a `fitz.IRect(x0, y0, x1, y1)`, `fitz.Point(x0, y0)` designates the top left corner of the pixmap. Note that the image area is **not initialized** and will contain crap data.

Parameters

- `colorspace` (*Colorspace*) – colorspace of the pixmap.
- `irect` (*IRect*) – Tte pixmap's area and location.

- `alpha` (*bool*) – Specifies whether transparency bytes should be included. Default is `False`.

`__init__`(*self*, *colorspace*, *source*[, *alpha*])

Copy and set colorspace: Copy `source` pixmap choosing the colorspace. Any colorspace combination is possible.

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.
- `alpha` (*bool*) – whether to also copy the source's alpha channel. If the source has no alpha, this parameter has no effect. If `False` the result will have no alpha.

`__init__`(*self*, *source*, *width*, *height*[, *clip*])

Copy and scale: Copy `source` pixmap choosing new width and height values. Supports partial copying.

Parameters

- `source` (`Pixmap`) – the source pixmap.
- `width` (*float*) – desired target width.
- `height` (*float*) – desired target height.
- `clip` (*IRect*) – a region of the source pixmap to take the copy from.

`__init__`(*self*, *source*)

Copy and add alpha: Identical copy from `source` with an added alpha channel. The alpha values are set to 255.

Parameters `source` (`Pixmap`) – the source pixmap, must not have alpha.

`__init__`(*self*, *filename*)

From a file: Create a pixmap from `filename`. Image type and all properties are determined automatically.

Parameters `filename` (*str*) – Path / name of the file. The origin of the resulting pixmap is `(0, 0)`.

`__init__`(*self*, *img*)

From memory: Create a pixmap from bytearray `img`. Image type and all properties are determined automatically.

Parameters `img` (*bytearray*) – Data containing a complete, valid image in one of the supported formats. Could have been created by something like `img = bytearray(open('somepic.png', 'rb').read())`. The origin of the resulting pixmap is (0,0). Type `bytes` is **not supported** here, because that cannot be distinguished from `string` in Python 2.

`__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 examples below.

Parameters

- `colorspace` (*Colorspace*) – Colorspace of the image. Together with `alpha` this parameter controls the interpretation of the `samples` area. The following must be true: `(colorspace.n + alpha) * width * height == len(samples)`.

- `width` (*int*) – image width
- `height` (*int*) – image height
- `samples` (*bytes*) – an area containing all pixels of the image. Must include alpha values if specified. Type `bytearray` is also supported.
- `alpha` (*bool*) – whether a transparency channel is included.

Caution: The method will not make a copy of `samples`, but rather record a pointer. Therefore make sure that it remains available throughout the lifetime of the pixmap. Otherwise the pixmap's image will likely be destroyed or even worse things will happen.

`__init__`(*self*, *doc*, *xref*)

From a PDF image: Create a pixmap from an image **contained in PDF** `doc` identified by its XREF number. All pimap properties are set by the image.

Parameters

- `doc` (*Document*) – an opened **PDF** document.
- `xref` (*int*) – the XREF number of the 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 always be set to 255 (non-transparent). If omitted, then all bytes including alpha are cleared to 0x00.
- `irect` (*IRect*) – 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 a value triple (red, green, blue). Only colorspaces *CS_GRAY* and *CS_RGB* are supported.

If the colorspace is *CS_GRAY*, `(red + green + blue)/3` will be taken as the tinting value.

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.

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.

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.

`setAlpha(`[*alphavalues*]`)`
Change the alpha values. The pixmap must have an alpha channel.

Parameters `alphavalues` (*bytes*) – the new alpha values. Type `bytearray` is also permitted. If provided, its length must be at least `width * height`. If omitted, alpha values are all set to 255 (no transparency).

`invertIRect(`*irect*`)`
Invert the color of all pixels in *IRect* `irect`.

Parameters `irect` (*IRect*) – The area to be inverted. Omit to invert everything.

`copyPixmap(`*source*, *irect*`)`
Copy the *IRect* part of `source` into the corresponding area of this one. The two pixmaps may have different dimensions and different colorspaces (provided each is either *CS_GRAY* or *CS_RGB*), but currently **must** have the same alpha property. 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. Then the corresponding data of this intersection are being copied. If the intersection is empty, nothing will happen.

If you want your `source` pixmap image to land at a specific target position, set its `x` and `y` attributes to the top left point of the desired rectangle before copying. See the example below for how this works.

Parameters

- `source` (*Pixmap*) – The pixmap from where to copy.
- `irect` (*IRect*) – The area to be copied.

`writeImage(`*filename*, *output="png"*`)`
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 ignored with a warning.

Parameters

- `filename` (*str*) – The filename to save to. Depending on the chosen output format, possible file extensions are `.pam`, `.pbm`, `.pgm`, `ppm`, `.pnm`, `.png` and `.tga`.
- `output` (*str*) – The requested image format. The default is `png` for which this function is equal to `writePNG()`, see below. Other possible values are `pam`, `pnm` and `tga`.

`writePNG(`*filename*`)`
Save the pixmap as a PNG file. Please note that only grayscale and RGB colorspaces are supported (this is **not** a MuPDF restriction). CMYK colorspaces must either be saved as `*.pam` files or be converted first.

Parameters `filename` (*str*) – The filename to save to (the extension `png` must be specified). Existing files will be overwritten without warning.

`getPNGData()`
Like `writePNG` but returnes a bytearray instead.

Return type bytearray

`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 `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 `alpha == 1`) transparency values for all pixels. `samples` is a memory area of size `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. See example 3.

Type bytes

`size`
Contains `len(pixmap)`. This will generally equal `len(pix.samples) + 60` (32bit systems, the delta is 88 on 64bit machines).

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.

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

4.11.1 Supported Input Image Types

The following file types are supported as input to construct pixmaps: **BMP, JPEG, GIF, TIFF, JXR,** and **PNG**. 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 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. In any case, this will turn the SVG into a raster image. If you need a **vector image** you must first convert it to a PDF and then display it e.g. via *`Page.showPDFpage()`*. There exist many tools for SVG-to-PDF conversion, among them the Python package svglib or Java solutions like Apache Batik. Have a look at our Wiki for example solutions.

4.11.2 Details on Saving Images with `writeImage()`

The following table shows possible combinations of file extensions, output formats and colorspaces of method `writeImage()`:

output =	CS_GRAY	CS_RGB	CS_CMYK
"pam"	.pam	.pam	.pam
"pnm"	.pnm, .pgm	.pnm, .ppm	invalid
"png"	.png	.png	invalid
"tga"	.tga	.tga	invalid

Note: Not all image file types are available, or at least common on all platforms, e.g. PAM is mostly 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 as a PNG.

4.11.3 Pixmap Example Code Snippets

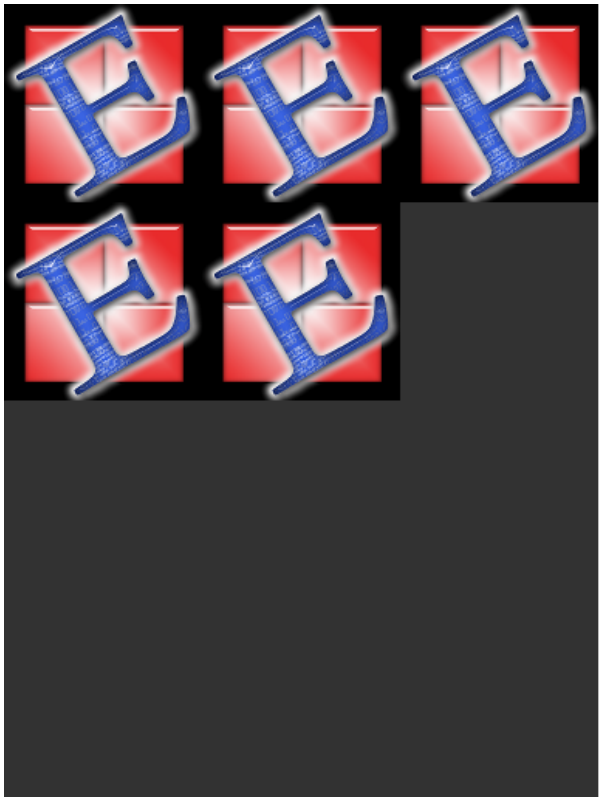
Example 1

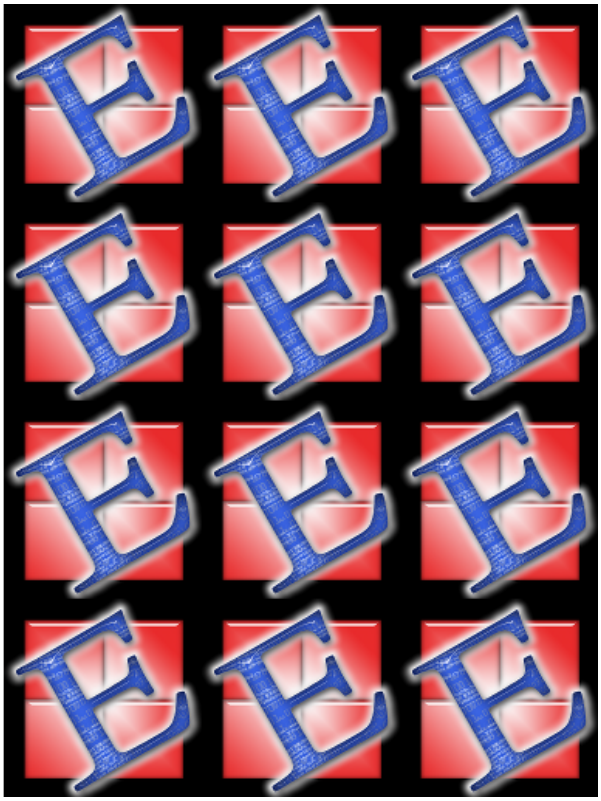
This shows how pixmaps can be used for purely graphical, non-PDF purposes. The script reads a PNG picture and creates a new PNG file which consist of 3 * 4 tiles of the original one:

```
import fitz
# create a pixmap of a picture
pix0 = fitz.Pixmap("editra.png")

# set target colorspace and pixmap dimensions and create it
tar_width  = pix0.width * 3              # 3 tiles per row
tar_height = pix0.height * 4             # 4 tiles per column
tar_irect  = fitz.IRect(0, 0, tar_width, tar_height)
# create empty target pixmap
tar_pix    = fitz.Pixmap(fitz.csRGB, tar_irect, pix0.alpha)
# clear target with a very lively stone-gray (thanks and R.I.P., Loriot)
tar_pix.clearWith(90)

# now fill target with 3 * 4 tiles of input picture
for i in range(4):
    pix0.y = i * pix0.height                    # modify input's y coord
    for j in range(3):
        pix0.x = j * pix0.width                 # modify input's x coord
        tar_pix.copyPixmap(pix0, pix0.irect)    # copy input to new loc
        # save all intermediate images to show what is happening
        fn = "target-%i-%i.png" % (i, j)
        tar_pix.writePNG(fn)
```

This is the input picture `editra.png` (taken from the wxPython directory `/tools/Editra/pixmaps`):

Here is the output, showing some intermediate picture and the final result:





Example 2

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")
```

Example 3

This shows how to interface with `PIL / Pillow` (the Python Imaging Library), thereby extending the reach of image files that can be processed:

```
>>> import fitz
>>> from PIL import Image
>>> pix = fitz.Pixmap(...)
>>> ...
>>> # create and save a PIL image
>>> img = Image.frombytes("RGB", [pix.width, pix.height], pix.samples)
>>> img.save(filename, 'jpeg')
>>> ...
>>> # opposite direction:
```

```
>>> # create a pixmap from any PIL-supported image file "some_image.xxx"
>>> img = Image.open("some_image.xxx").convert("RGB")
>>> samples = img.tobytes()
>>> pix = fitz.Pixmap(fitz.csRGB, img.size[0], img.size[1], samples, alpha=False)
```

4.12 Point

`Point` represents a point in the plane, defined by its x and y coordinates.

Attribute / Method	Short Description
Point.distance_to()	calculate distance to point or rect
Point.transform()	transform point with a matrix
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*, *list*)

Overloaded constructors.

Without parameters, `Point(0, 0)` will be created.

With another `point` specified, a **new copy** will be crated. A `list` must be Python sequence object of length 2. For a `list`, it is the user's responsibility to only provide numeric entries - **no error checking is done**, and invalid entries will receive a value of `-1.0`.

Parameters

- `x` (*float*) – X coordinate of the point
- `y` (*float*) – Y coordinate of the point

`distance_to`(*x*[, *unit*])

Calculates the distance to `x`, which may be a *Rect*, *IRect* or *Point*. The distance is given in units of either `px` (pixels, default), `in` (inches), `mm` (millimeters) or `cm` (centimeters).

Note: If `x` is a rectangle, the distance is calculated as if the rectangle were finite.

Parameters

- `x` (*Rect* or *IRect* or *Point*) – the object to which the distance is calculated.
- `unit` (*str*) – the unit to be measured in. One of `px`, `in`, `cm`, `mm`.

Returns distance to object `x`.
Return type float

`transform`(*m*)

Applies matrix `m` to the point.

Parameters `m` – The matrix to be applied.
Return type `Point`

`x`
`x Coordinate`

`y`
`y Coordinate`

4.12.1 Remark

A point's `p` attributes `x` and `y` can also be accessed as indices, e.g. `p.x == p[0]`, and the `tuple()` and `list()` functions yield sequence objects of its components.

4.12.2 Point Algebra

For a general background, see chapter *Operator Algebra for Geometry Objects*.

4.12.3 Examples

This should illustrate some basic uses:

```
>>> fitz.Point(1, 2) * fitz.Matrix(90)
fitz.Point(-2.0, 1.0)
>>>
>>> fitz.Point(1, 2) * 3
fitz.Point(3.0, 6.0)
>>>
>>> fitz.Point(1, 2) + 3
fitz.Point(4.0, 5.0)
>>>
>>> fitz.Point(25, 30) + fitz.Point(1, 2)
fitz.Point(26.0, 32.0)
>>> fitz.Point(25, 30) + (1, 2)
fitz.Point(26.0, 32.0)
>>>
>>> fitz.Point([1, 2])
fitz.Point(1.0, 2.0)
>>>
>>> -fitz.Point(1, 2)
fitz.Point(-1.0, -2.0)
>>>
>>> abs(fitz.Point(25, 30))
39.05124837953327
```

4.13 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. Their *Common Parameters* are however exported to a separate method, `finish()`. In addition, all draw methods return a *Point* object to support connected drawing paths. This point always equals the **"current point"**, that PDF maintains during path construction.

The class now also supports the text insertion methods `insertText()` and `insertTextbox()`. They need a slightly different handling compared to the draw methods (see below for details):

1. They do not use *`Shape.contents`*. Instead they directly modify *`Shape.totalcont`*.
2. They do not use nor need *`Shape.finish()`*.
3. They provide their own `color` and `morph` arguments.

4. They do not use nor change *Shape.lastPoint*.

As with the draw methods, text insertion requires using *Shape.commit()* to update the page.

Method / Attribute	Description
Shape.commit()	update the page's `/Contents` object
Shape.drawBezier()	draw a cubic Bézier curve
Shape.drawCircle()	draw a circle around a point
Shape.drawCurve()	draw a cubic Bézier using one helper point
Shape.drawLine()	draw a line
Shape.drawOval()	draw an ellipse
Shape.drawPolyline()	connect a sequence of points
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 draws
Shape.insertText()	insert text lines
Shape.insertTextbox()	insert text into a rectangle
Shape.contents	draw commands since last `finish()`
Shape.doc	stores the page's document
Shape.height	stores the page's height
Shape.lastPoint	stores the current point
Shape.page	stores the owning page
Shape.width	stores the page's width
Shape.totalcont	accumulated string to be stored in `/Contents`

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* objects `p1` to `p2`.

Parameters

- `p1` (*Point*) – starting point
- `p2` (*Point*) – end point

Return type *Point*

Returns the end point, `p2`.

`drawSquiggle`(*p1*, *p2*, *breadth = 2*)
Draw a squiggly (wavy, undulated) line from *Point* 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".

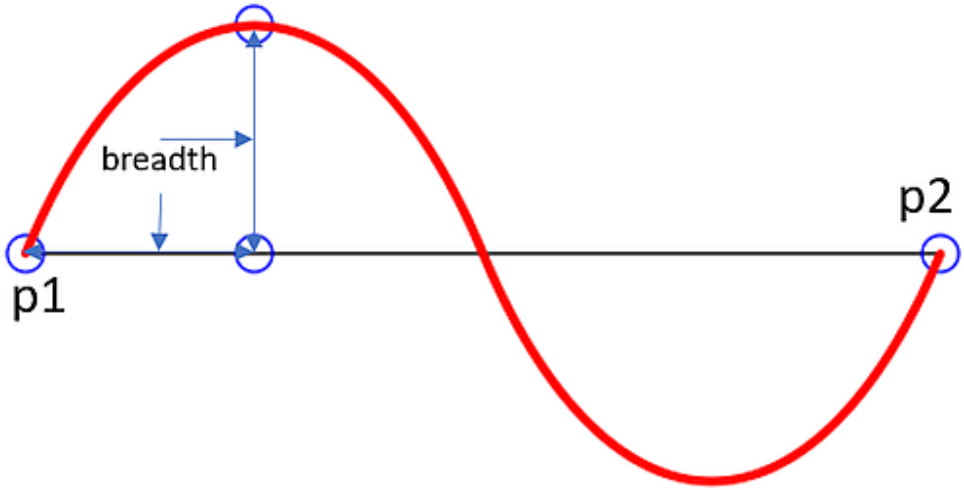
Parameters

- `p1` (*Point*) – starting point
- `p2` (*Point*) – end point

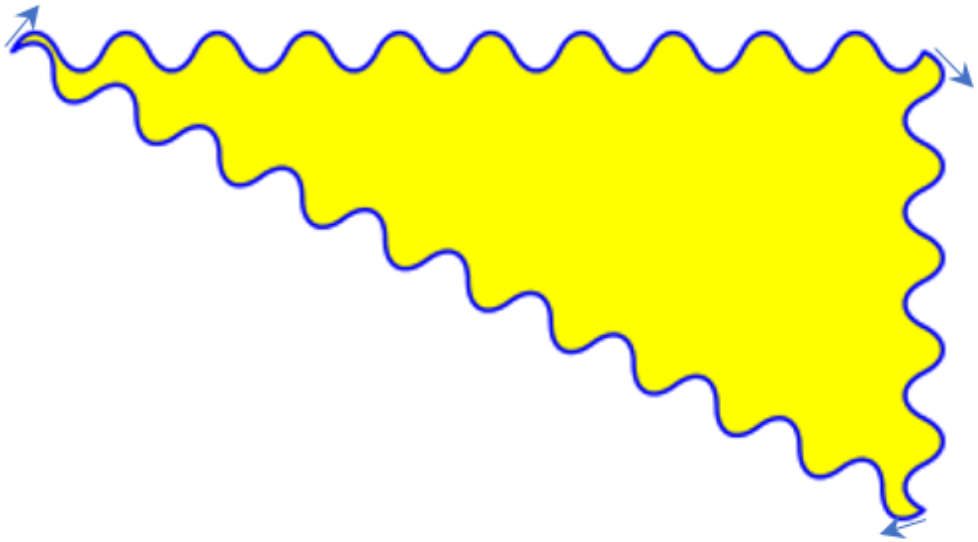
- `breadth` (*float*) – the amplitude of each wave. The condition `2 * breadth < abs(p2 - p1)` must be true to fit in at least one wave. See the following picture, which shows two points connected by one full period.

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.

`drawZigzag`(*p1*, *p2*, *breadth = 2*)

Draw a zigzag line from *Point* 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*) – starting point
- `p2` (*Point*) – 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* 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 Bézier curve from `p1` to `p4`, using `p2` and `p3` as control points.

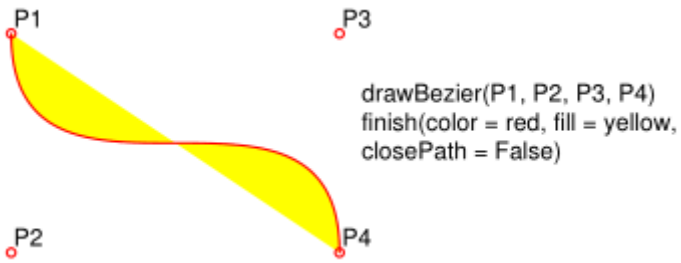
Parameters

- `p1` (*Point*) – starting point
- `p2` (*Point*) – control point 1
- `p3` (*Point*) – control point 2
- `p4` (*Point*) – end point

Return type *Point*

Returns the end point, `p4`.

Example:



`drawOval`(*rect*)

Draw an ellipse inside the given rectangle. If `rect` is a square, a standard circle is drawn. The drawing starts and ends at the middle point of the left rectangle side in a counter-clockwise movement.

Parameters `rect` (*Rect*) – rectangle, must be finite and not empty.

Return type *Point*

Returns the middle point of the left rectangle side.

`drawCircle`(*center*, *radius*)

Draw a circle given its center and radius. The drawing starts and ends at point `start = center - (radius, 0)` in a counter-clockwise movement. `start` corresponds to the middle point of the enclosing square's left border.

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*) – 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 Bézier 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 degress, then the resulting curve is a quarter ellipse (or quarter circle, if of same length) circumference.

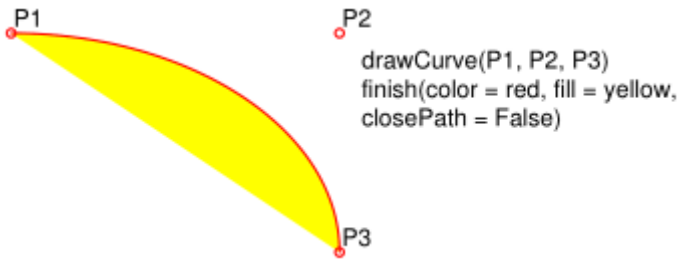
Parameters

- `p1` (*Point*) – starting point.
- `p2` (*Point*) – helper point.
- `p3` (*Point*) – end point.

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).

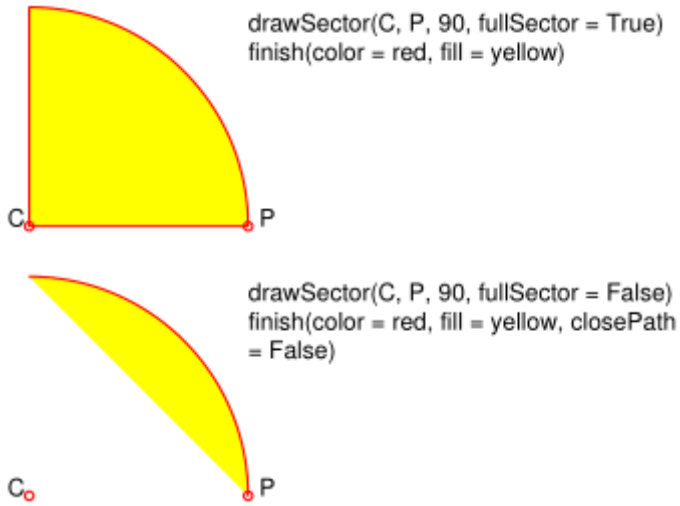
Parameters

- `center` (*Point*) – the center of the circle.
- `point` (*Point*) – 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 counter-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 a counter-clockwise movement.

Parameters `rect` (*Rect*) – where to put the rectangle on the page.

Return type *Point*

Returns `rect.top_left` (top-left corner of the rectangle).

`insertText`(*point*, *text*, *fontsize = 11*, *fontname = "Helvetica"*, *fontfile = None*, *idx = 0*, *set_simple = False*, *color = (0, 0, 0)*, *rotate = 0*, *morph = None*)

Insert text lines beginning at a *Point* `point`.

Parameters

- `point` (*Point*) – the bottom-left position of the first `text` character in pixels. `point.x` specifies the distance from left border, `point.y` the distance from top of page. This is independent from text orientation as requested by `rotate`. However, there must always be sufficient room "above", which can mean the distance from any of the four page borders.
- `text` (*str or 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. Only single byte character codes are currently supported.
- `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 counter-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* = *"Helvetica"*, *fontfile* = *None*, *idx* = *0*, *set_simple* = *False*, *color* = *(0, 0, 0)*, *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, depending on `rotate`. Line feeds will be respected as well as multiple spaces will be.

Parameters

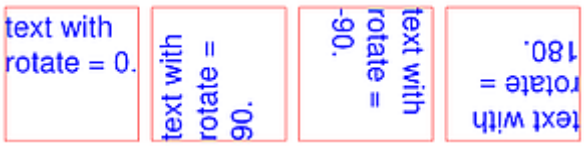
- `rect` (*Rect*) – the area to use. It must be finite and not empty.
- `buffer` – the text to be inserted. Must be specified as a string or a sequence of strings. Line breaks are respected also when occurring in a sequence entry.
- `align` (*int*) – align each text line. Default is 0 (left). Centered, 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.



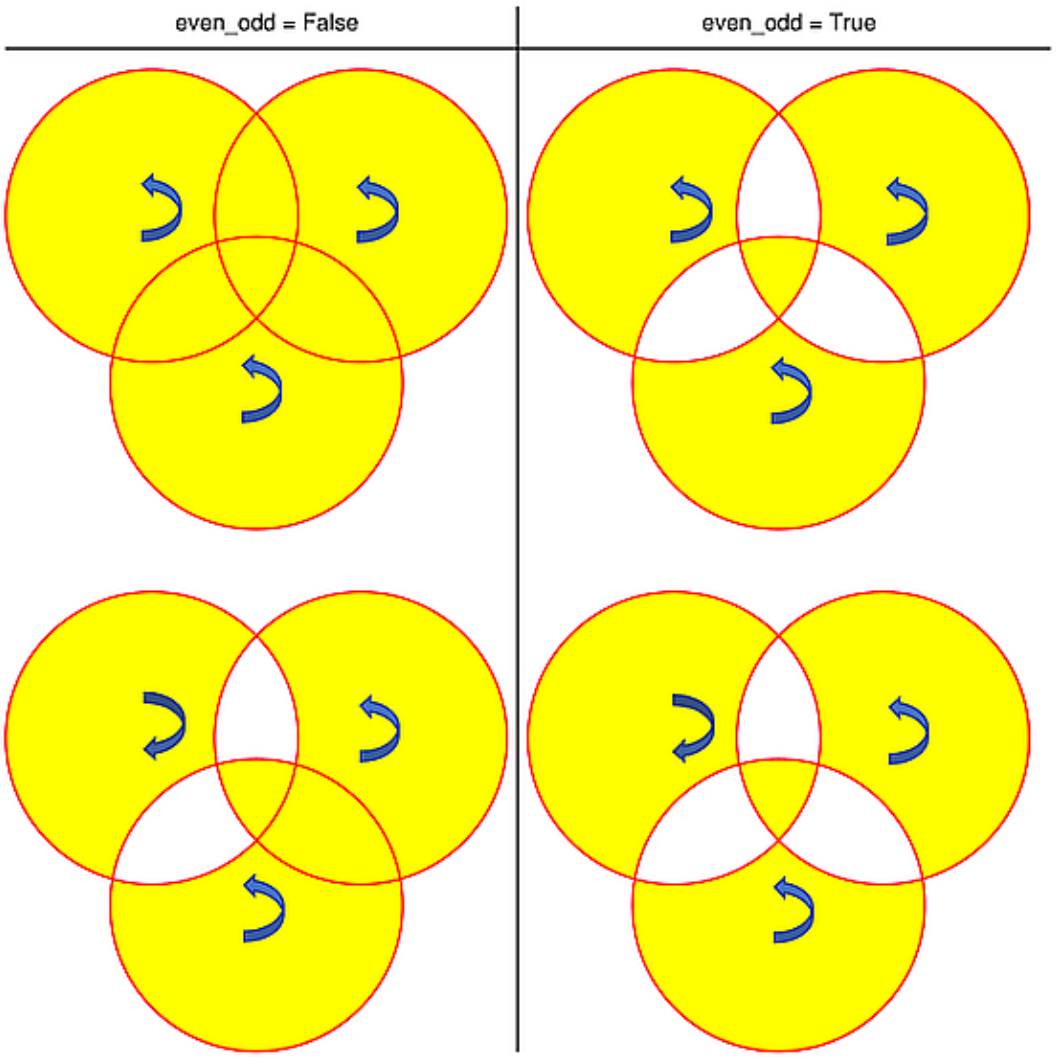
For a description of the other parameters see *Common Parameters*.

`finish`(*width = 1*, *color = (0, 0, 0)*, *fill = None*, *roundCap = True*, *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 compound drawing around some arbitrary pivotal *Point* `pivot` by applying *Matrix* `matrix` to it. 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: Method **"even-odd"** counts the number of overlaps of areas. Pixels in areas overlapping an odd number of times are regarded **inside**, otherwise **outside**. In contrast, the default method **"nonzero winding"** also looks at the area orientation: it counts `+1` if an

area is drawn counter-clockwise and `-1` else. If the result is zero,the pixel is regarded **outside**, otherwise **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 drawing commands. If a `Shape` is not committed, the page will not be changed. The method must be preceeded with at least one `finish()` or text insertion method.

Parameters `overlay` (*bool*) – determine whether to put the drawing in foreground (default) or background. Relevant only, if the page 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

`contents`
Accumulated command buffer for draw methods since last finish.

Type str

`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*

4.13.1 Usage

A drawing object is constructed by `img = 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:

```
>>> img = page.newShape()
>>> img.draw1(...)
>>> img.draw2(...)
>>> ...
>>> img.finish(width=..., color = ..., fill = ..., morph = ...)
>>> img.draw3(...)
>>> img.draw4(...)
>>> ...
```

```
>>> img.finish(width=..., color = ..., fill = ..., morph = ...)
>>> ...
>>> img.insertText*
>>> ...
>>> img.commit()
>>> ....
```

Notes

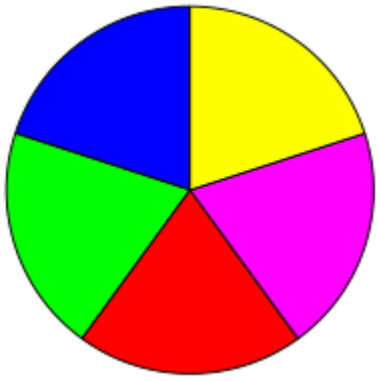
1. Each `finish()` combines the preceding draws into one logical shape, giving it common colors, line width, morphing, etc. If `closePath` is specified, it will also connect the end point of the last draw with the starting point of the first one.
2. To successfully create compound graphics, let each 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.contents* 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. With many draw / text operations, this will result in a much better performance, than issuing the corresponding page methods separately (they each do their own commit).

4.13.2 Examples

1. Create a full circle of pieces of pie in different colors.

```
>>> img  = 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 = img.drawSector(center, p0, beta,
                            fullSector = True) # draw piece
        # now fill it but do not connect ends of the arc
        img.finish(fill = cols[i], closePath = False)
>>> img.commit()                 # update the page
```

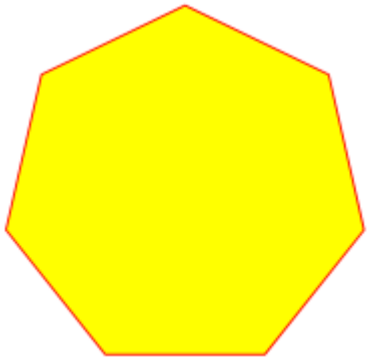
Here is an example for 5 colors:



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.

```
>>> img  = 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 = img.drawSector(center, p0, beta)
        points.append(p0)
>>> img.contents = ""             # do not draw the circle sectors
>>> img.drawPolyline(points)      # draw the polygon
>>> img.finish(color = (1,0,0), fill = (1,1,0), closePath = False)
>>> img.commit()
```

Here is the polygon for n = 7:



4.13.3 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.

---

idx (*int*)

Font files may contain more than one font. Use this parameter to select the right one. This setting cannot be reverted. Subsequent changes are ignored.

---

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 are always specified as RGB triples of floats from 0 to 1. To simplify color specification, method `getColor()` in `fitz.utils` may be used. 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*.

---

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.

roundCap (*bool*)

Cause lines, dashes and edges to be rounded (default). If false, sharp edges and square line and dashes ends will be generated. Rounded lines / dashes will end in a semi-circle with a diameter equal to line width and make longer by the radius of this semi-circle.

closePath (*bool*)

Causes the end point of a drawing to be automatically connected with the starting point (by a straight line).

4.14 Rect

`Rect` represents a rectangle defined by four floating point numbers x0, y0, x1, y1. They are viewed 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).
- 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.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.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*, *list*)

Overloaded constructors: `top_left`, `bottom_right` stand for *Point* objects, `list` is a Python sequence type with length 4, `rect` means another `Rect`, while the other parameters mean float coordinates. If `list` is specified, it is the user's responsibility to only provide numeric entries - **no error checking is done**, and invalid entries will receive a value of `-1.0`.

If `rect` is specified, the constructor creates a **new copy** of `rect`.

Without parameters, the 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!).

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
False
>>> r.round()
fitz.IRect(100, 100, 200, 100)
>>> r.round().isEmpty
True
```

To reproduce the effect on your platform, you may need to adjust the numbers a little.

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(`$\left[$*unit*$\right]$`)`

`getArea(`$\left[$*unit*$\right]$`)`

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 `r` (a `Rect` or *IRect*) have a non-empty rectangle in common. This will always be `False` if either is infinite or empty.

Parameters `r` (*IRect* or *Rect*) – the rectangle to check.

Return type bool

`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*

`width`
Contains the width of the rectangle. Equals `x1 - x0`.

Return type float

`height`
Contains the height of the rectangle. Equals `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

4.14.1 Remark

A rectangle's coordinates can also be accessed via index, e.g. `r.x0 == r[0]`, and the `tuple()` and `list()` functions yield sequence objects of its components.

4.14.2 Rect Algebra

For a general background, see chapter *Operator Algebra for Geometry Objects*.

4.14.3 Examples

Example 1 - different ways of construction:

```
>>> p1 = fitz.Point(10, 10)
>>> p2 = fitz.Point(300, 450)
>>>
>>> fitz.Rect(p1, p2)
fitz.Rect(10.0, 10.0, 300.0, 450.0)
>>>
>>> fitz.Rect(10, 10, 300, 450)
fitz.Rect(10.0, 10.0, 300.0, 450.0)
>>>
>>> fitz.Rect(10, 10, p2)
fitz.Rect(10.0, 10.0, 300.0, 450.0)
>>>
>>> fitz.Rect(p1, 300, 450)
fitz.Rect(10.0, 10.0, 300.0, 450.0)
```

Example 2 - what happens during rounding:

```
>>> r = fitz.Rect(0.5, -0.01, 123.88, 455.123456)
>>>
>>> r
fitz.Rect(0.5, -0.009999999776482582, 123.87999725341797, 455.1234436035156)
>>>
>>> r.round()      # = r.irect
fitz.IRect(0, -1, 124, 456)
```

Example 3 - inclusion and itersection:

```
>>> m = fitz.Matrix(45)
>>> r = fitz.Rect(10, 10, 410, 610)
>>> r * m
fitz.Rect(-424.2640686035156, 14.142135620117188, 282.84271240234375, 721.2489013671875)
>>>
>>> r | fitz.Point(5, 5)
fitz.Rect(5.0, 5.0, 410.0, 610.0)
>>>
```

```
>>> r + 5
fitz.Rect(15.0, 15.0, 415.0, 615.0)
>>>
>>> r & fitz.Rect(0, 0, 15, 15)
fitz.Rect(10.0, 10.0, 15.0, 15.0)
```

Example 4 - containment:

```
>>> r = fitz.Rect(...)      # any rectangle
>>> ir = r.irect            # its IRect version
>>> # even though you get ...
>>> ir in r
True
>>> # ... and ...
>>> r in ir
True
>>> # ... r and ir are still different types!
>>> r == ir
False
>>> # corners are always part of non-epmpty rectangles
>>> r.bottom_left in r
True
>>>
>>> # numbers are checked against coordinates
>>> r.x0 in r
True
```

Example 5 - create a finite copy:

Create a copy that is **guarantied to be finite** in two ways:

```
>>> r = fitz.Rect(...)      # any rectangle
>>>
>>> # alternative 1
>>> s = fitz.Rect(r.top_left, r.top_left)   # just a point
>>> s | r.bottom_right      # s is a finite rectangle!
>>>
>>> # alternative 2
>>> s = (+r).normalize()
>>> # r.normalize() changes r itself!
```

Example 6 - adding a Python sequence:

Enlarge rectangle by 5 pixels in every direction:

```
>>> r  = fitz.Rect(...)
>>> r1 = r + (-5, -5, 5, 5)
```

Example 7 - inline operations:

Replace a rectangle with its transformation by the inverse of a matrix-like object:

```
>>> r /= (1, 2, 3, 4, 5, 6)
```

CHAPTER

FIVE

OPERATOR ALGEBRA FOR GEOMETRY OBJECTS

Instances of classes *Point*, *IRect*, *Rect* and *Matrix* are collectively also called "geometry" objects.

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.

5.1 General Remarks

1. Operators can be either **binary** (i.e. involving two objects) or **unary**.
2. The result of binary operatorions is either a **new object** of the same class as the **left operand** or a bool.
3. The result of unary operations is either a bool, a float or the same object type.
4. All binary operators fully support in-place operations, i.e. if the operator is called "°", then something like `a °= b` is equivalent to `a = a ° b`.
5. The following binary operators are defined for all classes: `+, -, *, /`. They have a similar meaning as the corresponding numerical ones.
6. Rectangles have two additional binary operators `&, |`, details below.
7. For binary operations, the **second** operand may have a different type as the left one. Often, Python sequences (lists, tuples, arrays) are also allowed here. We allude to this fact by saying "point-like object" when we mean, that a *Point* is possible as well as a sequence of two numbers. Similar applies to "rect-like" (sequence length 4) or "matrix-like" (sequence length 6).

5.2 Unary Operations

- `bool(o)` - is false if and only if the components of `o` are all zero.
- `abs(o)` - is the Euclidean norm (square root of the sum of component squares) if `o` is a *Point* or a *Matrix*. For rectangles, the area is returned (result of `getArea()`).
- `+o` - is a copy of `o`.
- `-o` - is a copy of `o` with negated components.
- `~m` - is the inverse of *Matrix* `m`. The other geometry objects are not invertible w/r to multiplication.

5.3 Binary Operations

For the operators `+, -, *, /`, the **second operand** may be a number, which will be applied componentwise.

- `a + b`, `a - b` - component-wise execution, `b` must be `a`-like.
- `a * b`, `a / b` **- does the following for matrix-likes** `b`**:**
  - If `a` is a **point** or a **rectangle**, then `a.transform(b)`, resp. `a.transform(~b)` is executed.
  - If `a` is a **matrix**, then `a * b`, resp. `a * ~b` is executed.
- `a & b` - **intersection rectangle:** `a` must be a rectangle and `b` rect-like.
- `a | b` - **union rectangle:** `a` must be a rectangle, and `b` can be point-like or rect-like.
- `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 the result of `a.contains(b)` is returned.
- `a == b` - is true if `abs(a - b) == 0` and `type(a) == type(b)` (but maybe we have `id(a) != id(b)`).

CHAPTER

SIX

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.

6.1 Functions

The following are miscellaneous functions to be used by the experienced PDF programmer.

Function	Short Description
Document.FontInfos	PDF only: information on inserted fonts
Annot._cleanContents()	PDF only: clean the annot's `/Contents` objects
Annot._getXref()	PDF only: return XREF number of annotation
ConversionHeader()	return header string for `getText` methods
ConversionTrailer()	return trailer string for `getText` methods
Document._delXmlMetadata()	PDF only: remove XML metadata
Document._getGCTXerrmsg()	retrieve C-level exception message
Document._getNewXref()	PDF only: create and return a new XREF entry
Document._getObjectString()	PDF only: return object source code
Document._getOLRootNumber()	PDF only: return / create XREF of `/Outline`
Document._getPageObjNumber()	PDF only: return XREF and generation number of a page
Document._getPageXref()	PDF only: same as `_getPageObjNumber()`
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
Document._getXrefString()	PDF only: return object source code
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.getCharWidths()	PDF only: return a list of glyph widths of a font
Document.getPageRawText()	PDF only: return raw string between two points
getPDFnow()	return the current timestamp in PDF format
getPDFstr()	return PDF-compatible string
Page._cleanContents()	PDF only: clean the page's `/Contents` objects
Page._getContents()	PDF only: return a list of content numbers
Page._getXref()	PDF only: return XREF number of page
Page.getDisplayList()	create the page's display list
Page.extractTextLines()	return text between two points
Page.extractTextRect()	return text inside a rectangle
Page.insertFont()	PDF only: store a new font in the document
Page.run()	run a page through a device
PaperSize()	return width, height for known 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 insensitive), optionally suffixed with "-L" (landscape) or "-P" (portrait). No suffix defaults to portrait.

Parameters `s` (*str*) – a format name like `"A4"` or `"letter-l"`.

Return type tuple

Returns `(width, height)` of the paper format. For an unknown format `(-1, -1)` is returned. Esamples: `PaperSize("A4")` returns `(595, 842)` and `PaperSize("letter-l")` delivers `(792, 612)`.

`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.

`getPDFstr(`*obj*, *brackets = True*`)`

Make a PDF-compatible string: if `obj` contains code points `ord(c) > 255`, then it will be converted to UTF-16BE as a hexadecimal character string like `<feff...>`. Otherwise, if `brackets = True`, it will enclose the argument in `()` replacing any characters with code points `ord(c) > 127` by their octal number `\nnn` prefixed with a backslash. If `brackets = False`, then the string is returned unchanged.

Parameters `obj` (*str or bytes or unicode*) – the object to convert

Return type str

Returns PDF-compatible string enclosed in either `()` or `<>`.

`ConversionHeader(`*output = "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._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._getXmlMetadataXref()`
Return he XML-based metadata object id from 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.

`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]`.

`Page._getXref()`
Page version for `_getPageObjNumber()` only delivering the XREF (not the generation number).

`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.insertFont(fontname = "Helvetica", fontfile = None, idx = 0, set_simple = False)`
Store a new font for the page and return its XREF. If the page already references this font, it is a no-operation and just the XREF is returned.

Parameters

- `fontname` (*str*) – The reference name of the font. If the name does not occur in *Page.getFontList()*, then this must be either the name of one of the *PDF Base 14 Fonts*, or `fontfile` must also be given. Following this method, font name prefixed with a slash "/" can be used to refer to the font in text insertions. If it appears in the list, the method ignores all other parameters and exits with the xref number.
- `fontfile` (*str*) – font file name. This file will be embedded in the PDF.

- `idx` (*int*) – index of the font in the given file. Has no meaning and is ingored if `fontfile` is not specified. Default is zero. An invalid index will cause an exception.

  Note: Certain font files can contain more than one font. This parameter can be used to select the right one. PyMuPDF has no way to tell whether the font file indeed contains a font for any non-zero index.

  Caution: Only the first choice of `idx` will be honored - subsequent specifications are ignored.

- `set_simple` (*bool*) – When inserting from a font file, a "Type0" font will be installed by default. This option causes the font to be installed as a simple font instead. Only 1-byte characters will then be presented correctly, others will appear as "?" (question mark).

  Caution: Only the first choice of `set_simple` will be honored. Subsequent specifications are ignored.

Return type int

Returns

the XREF of the font. PyMuPDF records inserted fonts in two places:

1. An inserted font will appear in *Page.getFontList()*.
2. *Document.FontInfos* records information about all fonts that have been inserted by this method on a document-wide basis.

`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 belongig to the page. The length of this list will always be at least one.

Return type list

Returns a list of XREF integers.

Each page has one or more associated contents objects (streams) which contain PDF operator syntax describing what appears where 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 XREF 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[1][2].

[1] If a page has multiple contents streams, they are treated as being one logical stream when the document is processed by reader software. A single operator cannot be split between stream boundaries, but a single **instruction** may well be. E.g. invoking the display of an image looks like this: `q a b c d e f cm /imageid Do Q`. Any single of these items (PDF notation: "lexical tokens") is always contained in one stream, but `q a b c d e f cm` may be in one and `/imageid Do Q` in the next one.

[2] Note that `/Contents` objects (similar to /Resources) may be **shared** among pages. A change to a contents stream

---

`Page._cleanContents()`
Clean all `/Contents` objects associated with this page (including contents of all annotations). "Cleaning" includes syntactical corrections, standardizations and "pretty printing" of the contents stream. If a page has several contents objects, they will be combined into one. Any discrepancies between `/Contents` and `/Resources` objects are also resolved / corrected. Note that the resulting contents stream will be stored uncompressed (if you do not specify `deflate` on save). See *Page._getContents()* for more details.

Return type int

Returns 0 on success.

`Annot._getXref()`
Return the xref number of an annotation.

Return type int

Returns XREF number of the annotation.

`Annot._cleanContents()`
Clean the `/Contents` streams associated with the annotation. This is the same type of action *Page._cleanContents()* performs - just restricted to this annotation.

Return type int

Returns 0 if successful (exception raised otherwise).

`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 fontsize 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`:

may therefore affect other pages, too. To avoid this: (1) use *Page._cleanContents()*, (2) read the `/Contents` object (there will now be only one left), (3) make your changes.

```
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.getPageRawText(`*pno*, *p1*, *p2*`)`
Return lines of raw text contained between a pair of points.

Parameters

- `pno` (*int*) – page number.
- `p1` (*Point*) – Text delimiter point.
- `p2` (*Point*) – Text delimiter point.

Return type string

Returns see the page version of this mehod.

`Page.extractTextLines(`*p1*, *p2*`)`
Return lines of text contained between a pair of points.

Parameters

- `p1` (*Point*) – text delimiter point.
- `p2` (*Point*) – text delimiter point.

Return type str

Returns text lines between the two points (UTF-8 encoded).

`Page.extractTextRect(`*rect*`)`
Return lines of text contained in a rectangle.

Parameters `rect` (*Rect*) – rectangle.

Return type str

Returns text occurring inside the rectangle.

`Document._getObjectString(`*xref*`)`

`Document._getXrefString(`*xref*`)`
Return the string ("source code") representing an arbitrary object. For stream objects, only the non-stream part is returned. To get the stream content, use *_getXrefStream()*.

Parameters `xref` (*int*) – XREF number.

Return type string

Returns the string defining the object identified by `xref`.

`Document._getGCTXerrmsg()`
Retrieve exception message text issued by PyMuPDF's low-level code. This in most cases, but not always, are MuPDF messages. This string will never be cleared - only overwritten as needed. Only rely on it if a `RuntimeError` had been raised.

Return type str

Returns last C-level error message on occasion of a `RuntimeError` exception.

`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 the XREF number `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 decompressed content stream of the object referenced by `xref`. If the object has / is no stream, an exception is raised.

Parameters `xref` (*int*) – XREF number.

Return type str or bytes

Returns the (decompressed) stream of the object. This is a string in Python 2 and a `bytes` object in Python 3.

`Document._updateStream(`*xref*, *stream*`)`
Replace the stream of an object identified by `xref`. If the object has no stream, an exception is raised. The function automatically performs a compress operation ("deflate").

Parameters

- `xref` (*int*) – XREF number.
- `stream` (*bytes or bytearray*) – the new content of the stream.

Return type int

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[2] 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[3] 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.extractFont(`*xref*, *info_only = False*`)`

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).

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, saves 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()
```

Caution: The basename is returned unchanged from the PDF. So it may contain characters (such as blanks) which disqualify it as a valid filename for your operating system. Take appropriate action.

[3] Resources objects are inheritable. This means that many pages can share one. Keeping a page's `/Resources` object in sync with changes of its `/Contents` therefore may require creating an own `/Resources` object for the page. This can best be achieved by using `clean` when saving, or by invoking *`Page._cleanContents()`*.

`Document.FontInfos`

Contains following information for any font inserted via *Page.insertFont()*:

- 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

6.2 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.

6.3 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 = TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE*)
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 `TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE = 3`, i.e. ligatures are **passed through** (not decomposed into components), 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*

6.4 TextPage

`TextPage` represents the text of a page.

Method	Short Description
TextPage.extractText()	Extract the page's plain text
TextPage.extractTEXT()	synonym of previous
TextPage.extractHTML()	Extract the page's text in HTML format
TextPage.extractJSON()	Extract the page's text in JSON format
TextPage.extractXHTML()	Extract the page's text in XHTML format
TextPage.extractXML()	Extract the page's text in XML format
TextPage.search()	Search for a string in the page

Class API

`class TextPage`

`extractText()`

`extractTEXT()`
Extract the text from a `TextPage` object. Returns a string of the page's complete text. The text is UTF-8 unicode and in the same sequence as the PDF creator specified it. If this looks awkward for your document, consider using a program that re-arranges the text according to a more familiar layout, e.g. PDF2TextJS.py in the examples directory. Or use another extraction method which also provides text position information like *TextPage.extractHTML()*, *TextPage.extractXML()*, or `Page.extractTextList()`.

Return type str

`extractHTML()`
Extract all text and images in HTML format. This version contains complete formatting and positioning information on line level. Images will be included as base64 strings. You need a HTML package to interpret the output. Also see *Controlling Quality of HTML Output*.

Return type str

`extractJSON()`
Extract all text in JSON format. Provides same information detail as HTML (including images). You need a JSON module to interpret the output. The result will be nested Python dictionaries and lists. See below for the structure.

Return type str

`extractXHTML()`
Extract all text in XHTML format. Text information detail is comparable with `extractTEXT`, but also contains images. This method makes no attempt to re-create the original visual appearance.

Return type str

`extractXML()`
Extract all text in XML format. This contains complete formatting information about every single character on the page: font, size, line, paragraph, location, etc. Contains no images.

Return type str

`search(`*string*, *hit_ max = 16*`)`
Search for `string`.

Parameters

- `string` (*str*) – The string to search for.
- `hit_max` (*int*) – Maximum number of expected hits (default 16).

Return type list

Returns a list of *Rect* objects (without transformation), each surrounding a found `string` occurrence.

Note: All of the above can be achieved by using the appropriate *Page.getText()* and *Page.searchFor()* methods. Also see further down and in the *Page* chapter for examples on how to create a valid file format by adding respective headers and trailers.

6.4.1 Structure of `TextPage.extractJSON()`

A text page in JSON format is a nested object consisting of dictionaries and lists.

Page Dictionary

Key	Value
width	page width in pixels *(float)*
height	page height in pixels *(float)*
blocks	list of blocks *(list)*

Block Dictionaries

Blocks come in two types with a different structure: image blocks and text blocks.

Image block:

Key	Value
type	1 = image *(int)*
bbox	block / image rectangle, formatted as `list(fitz.Rect)`
imgtype	image type *(int)*, see list below
width	original image width *(float)*
height	original image height *(float)*
image	image content *(base64 str)*, may be `None`

Image type values:

- 0 (unknown): image type could not be determined and is provided as PNG if possible
- 1 (raw): uncompressed samples
- 2 (FAX)
- 3 (flate)
- 4 (LZW)
- 5 (RLD)
- 6 (BMP)
- 7 (GIF)
- 8 (JPEG)
- 9 (JPX)
- 10 (JXR)
- 11 (PNG)
- 12 (PNM)
- 13 (TIFF)

Text block:

Key	Value
type	0 = text *(int)*
bbox	block rectangle, formatted as `list(fitz.Rect)`
lines	list of text lines *(list)*

Line Dictionary

Key	Value
bbox	line rectangle, formatted as `list(fitz.Rect)`
wmode	writing mode *(int)*: 0 = horizontal, 1 = vertical
dir	writing direction *(tuple of floats)*: `[x, y]`
spans	list of spans *(list)*

The entries of writing direction `dir` 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.

Span Dictionary

Spans contain the actual text. In contrast to MuPDF versions up to 1.11, a span no longer includes positioning information. Therefore, to reconstruct the text a line, span text pieces must be concatenated. A span now contains font information. A line contains more than one span only, when any changes of the font or its attributes occur.

Key	Value
font	name of font *(str)*
size	font size *(float)*
flags	font characteristics *(int)*
text	text *(str)*

`flags` is a set of bools describing the font:

- bit 0: superscripted text
- bit 1: italic
- bit 2: serifed
- bit 3: monospaced
- bit 4: bold

6.4.2 Full Document Output in JSON Format

Converting a document to JSON format requires a little programmer attention. Use the following schema to create a valid (i.e. de-serializable JSON) document:

```
>>> doc = fitz.open(...)     # maybe any document type!
>>> jsonfile = open("document.json", "w")
>>> pno = 0
>>> jsonfile.write(fitz.ConversionHeader("json", filename = doc.name))
>>> for page in doc:
        if pno > 0:
            jsonfile.write(",\n")    # comma needed between pages!
        jsonfile.write(page.getText("json"))
        pno += 1
>>> jsonfile.write(fitz.ConversionTrailer("json"))
>>> jsonfile.close()
```

The document level dictionary then looks like so:

Key	Value
document	specified filename *(str)*
pages	list of pages *(list)*

6.5 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.

6.5.1 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, or you store it when a page is visited for the first time.

Note, that for everything what follows, only the display list is needed - the corresponding *Page* object could have been deleted.

6.5.2 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 20% up to 50% shorter than that of `Page.getPixMap()`.

6.5.3 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 found locations:
        pix.invertIRect(r.irect)              # e.g. invert colors in rectangle
```

6.5.4 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 value 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
```

6.5.5 Further Performance improvements

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).

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 (memory and disk space) if the document is graphics oriented.

CHAPTER

SEVEN

CONSTANTS AND ENUMERATIONS

Constants and enumerations of MuPDF as implemented by PyMuPDF. Each of the following variables is accessible as `fitz.variable`.

7.1 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

7.2 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	one of the *PDF Base 14 Fonts* (cannot be extracted)

7.3 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.

7.4 Preserve Text Flags

Options controlling the amount of data a text device parses into a *TextPage*.

`TEXT_PRESERVE_LIGATURES`
1 - If this option is activated ligatures are passed through to the application in their original form. If this option is deactivated 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 this option is activated whitespace is passed through to the application in its original form. If this option is deactivated any type of horizontal whitespace (including horizontal tabs) will be replaced with space characters of variable width.

`TEXT_PRESERVE_IMAGES`
4 - If this option is set, then images will be stored in the structured text structure. The default is to ignore all images.

7.5 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

7.6 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

7.7 Annotation Types

Possible values (integer) for PDF annotation types. See chapter 8.4.5, pp. 615 of the Adobe manual for more details.

ANNOT_TEXT
0 - Text annotation

ANNOT_LINK
1 - Link annotation

ANNOT_FREETEXT
2 - Free text annotation

ANNOT_LINE
3 - Line annotation

ANNOT_SQUARE
4 - Square annotation

ANNOT_CIRCLE
5 - Circle annotation

ANNOT_POLYGON
6 - Polygon annotation

ANNOT_POLYLINE
7 - PolyLine annotation

ANNOT_HIGHLIGHT
8 - Highlight annotation

ANNOT_UNDERLINE
9 - Underline annotation

ANNOT_SQUIGGLY
10 - Squiggly-underline annotation

ANNOT_STRIKEOUT
11 - Strikeout annotation

ANNOT_STAMP
12 - Rubber stamp annotation

ANNOT_CARET
13 - Caret annotation

ANNOT_INK
14 - Ink annotation

ANNOT_POPUP
15 - Pop-up annotation

ANNOT_FILEATTACHMENT
16 - File attachment annotation

ANNOT_SOUND
17 - Sound annotation

ANNOT_MOVIE
18 - Movie annotation

`ANNOT_WIDGET`
19 - Widget annotation

`ANNOT_SCREEN`
20 - Screen annotation

`ANNOT_PRINTERMARK`
21 - Printers mark annotation

`ANNOT_TRAPNET`
22 - Trap network annotation

`ANNOT_WATERMARK`
23 - Watermark annotation

`ANNOT_3D`
24 - 3D annotation

7.8 Annotation Flags

Possible mask values for PDF annotation flags.

Note: Annotation flags is a bit field, so test the truth of its bits with the `&` operator. When changing flags for an annotation, use the `|` operator to combine several values. The following descriptions were extracted from the Adobe manual, pages 608 pp.

`ANNOT_XF_Invisible`
1 - If set, do not display the annotation if it does not belong to one of the standard annotation types and no annotation handler is available. If clear, display such an unknown annotation using an appearance stream specified by its appearance dictionary, if any.

`ANNOT_XF_Hidden`
2 - If set, do not display or print the annotation or allow it to interact with the user, regardless of its annotation type or whether an annotation handler is available. In cases where screen space is limited, the ability to hide and show annotations selectively can be used in combination with appearance streams to display auxiliary pop-up information similar in function to online help systems.

`ANNOT_XF_Print`
4 - If set, print the annotation when the page is printed. If clear, never print the annotation, regardless of whether it is displayed on the screen. This can be useful, for example, for annotations representing interactive pushbuttons, which would serve no meaningful purpose on the printed page.

`ANNOT_XF_NoZoom`
8 - If set, do not scale the annotation's appearance to match the magnification of the page. The location of the annotation on the page (defined by the upper-left corner of its annotation rectangle) remains fixed, regardless of the page magnification.

`ANNOT_XF_NoRotate`
16 - If set, do not rotate the annotation's appearance to match the rotation of the page. The upper-left corner of the annotation rectangle remains in a fixed location on the page, regardless of the page rotation.

`ANNOT_XF_NoView`
32 - If set, do not display the annotation on the screen or allow it to interact with the user. The annotation may be printed (depending on the setting of the Print flag) but should be considered hidden for purposes of on-screen display and user interaction.

`ANNOT_XF_ReadOnly`
64 - If set, do not allow the annotation to interact with the user. The annotation may be displayed

or printed (depending on the settings of the NoView and Print flags) but should not respond to mouse clicks or change its appearance in response to mouse motions.

`ANNOT_XF_Locked`
128 - If set, do not allow the annotation to be deleted or its properties (including position and size) to be modified by the user. However, this flag does not restrict changes to the annotation's contents, such as the value of a form field.

`ANNOT_XF_ToggleNoView`
256 - If set, invert the interpretation of the NoView flag for certain events. A typical use is to have an annotation that appears only when a mouse cursor is held over it.

`ANNOT_XF_LockedContents`
512 - If set, do not allow the contents of the annotation to be modified by the user. This flag does not restrict deletion of the annotation or changes to other annotation properties, such as position and size.

7.9 Annotation Line End Styles

The following descriptions are taken from the Adobe manual TABLE 8.27 on page 630.

`ANNOT_LE_None`
0 - No line ending.

`ANNOT_LE_Square`
1 - A square filled with the annotation's interior color, if any.

`ANNOT_LE_Circle`
2 - A circle filled with the annotation's interior color, if any.

`ANNOT_LE_Diamond`
3 - A diamond shape filled with the annotation's interior color, if any.

`ANNOT_LE_OpenArrow`
4 - Two short lines meeting in an acute angle to form an open arrowhead.

`ANNOT_LE_ClosedArrow`
5 - Two short lines meeting in an acute angle as in the OpenArrow style (see above) and connected by a third line to form a triangular closed arrowhead filled with the annotation's interior color, if any.

`ANNOT_LE_Butt`
6 - (PDF 1.5) A short line at the endpoint perpendicular to the line itself.

`ANNOT_LE_ROpenArrow`
7 - (PDF 1.5) Two short lines in the reverse direction from OpenArrow.

`ANNOT_LE_RClosedArrow`
8 - (PDF 1.5) A triangular closed arrowhead in the reverse direction from ClosedArrow.

`ANNOT_LE_Slash`
9 - (PDF 1.6) A short line at the endpoint approximately 30 degrees clockwise from perpendicular to the line itself.

CHAPTER

EIGHT

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 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.

8.1 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)] ...
```

8.2 Printing the Color Database

If you want to actually see how the many available colors look like, use scripts colordbRGB.py or colordbHSV.py 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.


hotpink3
hotpink3
violetred1
violetred1
violetred2
violetred2
violetred3
violetred3
violetred4
violetred4
deeppink1
deeppink1
deeppink2
deeppink2
deeppink3
deeppink3
deeppink4
deeppink4
mediumvioletred
mediumvioletred
orchid2
orchid2
orchid1
orchid1
orchid
orchid
orchid3
orchid3
orchid4
orchid4
magenta4
magenta4
violet
violet
plum
plum
plum1
plum1
plum2
plum2
thistle3
thistle3
thistle4
thistle4
mediumorchid1
mediumorchid1
mediumorchid2
mediumorchid2
mediumorchid
mediumorchid

CHAPTER

NINE

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.

9.1 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, 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):

Runtime	Tool			
File	1 fitz	2 pdfrw	3 pdftk	4 PyPDF2
Adobe.pdf	5,25	21,06	112,39	692,23
Evolution.pdf	0,16	0,46	1,05	0,89
PyMuPDF.pdf	0,04	0,19	0,82	0,88
sdw_2015_01.pdf	0,23	1,23	5,41	6,45
sdw_2015_02.pdf	0,29	1,52	7,05	6,70
sdw_2015_03.pdf	0,51	2,77	11,49	11,98
sdw_2015_04.pdf	0,31	2,15	7,44	7,21
sdw_2015_05.pdf	0,35	1,69	7,60	7,59
sdw_2015_06.pdf	0,75	3,31	13,97	14,54
sdw_2015_07.pdf	0,37	2,11	10,17	9,72
sdw_2015_08.pdf	0,46	1,94	8,80	8,69
sdw_2015_09.pdf	0,79	2,35	10,58	10,42
sdw_2015_10.pdf	0,36	1,88	3,53	6,64
sdw_2015_11.pdf	2,41	12,69	37,12	60,40
sdw_2015_12.pdf	0,51	2,19	9,25	10,03
Gesamtergebnis	**12,78**	**57,54**	**246,66**	**854,36**

1,00	4,50	19,30	66,85
	1,00	4,29	14,85
		1,00	3,46

If we leave out the Adobe manual, this table looks like

Runtime	Tool			
File	1 fitz	2 pdfrw	3 pdftk	4 PyPDF2
Evolution.pdf	0,16	0,46	1,05	0,89
PyMuPDF.pdf	0,04	0,19	0,82	0,88
sdw_2015_01.pdf	0,23	1,23	5,41	6,45
sdw_2015_02.pdf	0,29	1,52	7,05	6,70
sdw_2015_03.pdf	0,51	2,77	11,49	11,98
sdw_2015_04.pdf	0,31	2,15	7,44	7,21
sdw_2015_05.pdf	0,35	1,69	7,60	7,59
sdw_2015_06.pdf	0,75	3,31	13,97	14,54
sdw_2015_07.pdf	0,37	2,11	10,17	9,72
sdw_2015_08.pdf	0,46	1,94	8,80	8,69
sdw_2015_09.pdf	0,79	2,35	10,58	10,42
sdw_2015_10.pdf	0,36	1,88	3,53	6,64
sdw_2015_11.pdf	2,41	12,69	37,12	60,40
sdw_2015_12.pdf	0,51	2,19	9,25	10,03
Gesamtergebnis	**7,53**	**36,48**	**134,28**	**162,13**

1,00	4,84	17,82	21,52
	1,00	3,68	4,44
		1,00	1,21

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.

9.2 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 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)
- **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 site.

9.3 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):

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

TEN

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)?

10.1 General structure of a TextPage

Information contained in a *TextPage* has the following hierarchy:

```
<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 font information and characters that share a common baseline.

10.2 Plain Text

This function 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:

```
PyMuPDF Documentation
Release 1.12.0
Jorj X. McKie
Dec 04, 2017
```

10.3 HTML

HTML output fully reflects the structure of the page's `TextPage` - much like JSON below. This includes images, font information and text positions. If wrapped in HTML header and trailer code, it can readily be displayed be a browser. Our above example:

```
<div style="width:595pt;height:841pt">
<img style="top:88pt;left:327pt;width:195pt;height:86pt" src="
/9j/4AAQSkZJRgABAQEAYABgAAD/4Q (... omitted image data ...) ">
<p style="top:189pt;left:195pt;"><b><span style="font-family:SFSX2488,serif;font-size:24.
↪7871pt;">PyMuPDF Documentation</span></b></p>
<p style="top:223pt;left:404pt;"><b><i><span style="font-family:SFSO1728,serif;font-size:17.
↪2154pt;">Release 1.12.0</span></i></b></p>
<p style="top:371pt;left:400pt;"><b><span style="font-family:SFSX1728,serif;font-size:17.
↪2154pt;">Jorj X. McKie</span></b></p>
<p style="top:637pt;left:448pt;"><b><span style="font-family:SFSX1200,serif;font-size:11.
↪9552pt;">Dec 04, 2017</span></b></p>
</div>
```

10.4 Controlling Quality of HTML Output

Though HTML output has improved a lot in MuPDF v1.12.0, it currently 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
        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!")
```

10.5 JSON

JSON 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). Have a look at PDF2textJS.py. Images are stored base64 encoded. Here is how this looks like:


```
{"width": 595.276, "height": 841.89,
 "blocks": [
  {"type": 1, "bbox": [327.526, 88.936, 523.276, 175.186],
   "imgtype": 8, "width": 261, "height": 115, "image":
"/9j/4AAQSkZJRgABAQEAYABgAAD/4QBmRXhpZgA (... omitted image data ...) "
  },
  {"type": 0, "bbox": [195.483, 189.041, 523.243, 218.91],
   "lines": [
    {"bbox": [195.483, 189.041, 523.243, 218.91], "wmode": 0, "dir": [1, 0],
     "spans": [
      {"font": "SFSX2488", "size": 24.7871, "flags": 20, "text": "PyMuPDF Documentation"}
     ]
    }
   ]
  },
  {"type": 0, "bbox": [404.002, 223.505, 523.305, 244.49],
   "lines": [
    {"bbox": [404.002, 223.505, 523.305, 244.49], "wmode": 0, "dir": [1, 0],
     "spans": [
      {"font": "SFSO1728", "size": 17.2154, "flags": 22, "text": "Release 1.12.0"}
     ]
    }
   ]
  },
  {"type": 0, "bbox": [400.529, 371.31, 517.284, 392.312],
   "lines": [
    {"bbox": [400.529, 371.31, 517.284, 392.312], "wmode": 0, "dir": [1, 0],
     "spans": [
      {"font": "SFSX1728", "size": 17.2154, "flags": 20, "text": "Jorj X. McKie"}
     ]
    }
   ]
  },
  {"type": 0, "bbox": [448.484, 637.531, 523.252, 652.403],
   "lines": [
    {"bbox": [448.484, 637.531, 523.252, 652.403], "wmode": 0, "dir": [1, 0],
     "spans": [
      {"font": "SFSX1200", "size": 11.9552, "flags": 20, "text": "Dec 13, 2017"}
     ]
    }
   ]
  }
 ]
}
```

10.6 XML

The XML version takes the level of detail even a lot deeper: every single character is provided with its position detail, and every span also contains font information:

```
<page width="595.276" height="841.89">
<image bbox="327.526 88.936038 523.276 175.18604" />
<block bbox="195.483 189.04106 523.2428 218.90952">
<line bbox="195.483 189.04106 523.2428 218.90952" wmode="0" dir="1 0">
<font name="SFSX2488" size="24.7871">
<char bbox="195.483 189.04106 214.19727 218.90952" x="195.483" y="211.052" c="P"/>
<char bbox="214.19727 189.04106 227.75582 218.90952" x="214.19727" y="211.052" c="y"/>
<char bbox="227.75582 189.04106 253.18738 218.90952" x="227.75582" y="211.052" c="M"/>
<char bbox="253.18738 189.04106 268.3571 218.90952" x="253.18738" y="211.052" c="u"/>
(... omitted data ...)
</font>
</line>
</block>
<block bbox="404.002 223.5048 523.30477 244.49039">
<line bbox="404.002 223.5048 523.30477 244.49039" wmode="0" dir="1 0">
<font name="SFSO1728" size="17.2154">
<char bbox="404.002 223.5048 416.91358 244.49039" x="404.002" y="238.94702" c="R"/>
(... omitted data ...)
<char bbox="513.33706 223.5048 523.30477 244.49039" x="513.33706" y="238.94702" c="0"/>
</font>
</line>
</block>
(... omitted data ...)
</page>
```

We have successfully tested `lxml` to interpret this output.

10.7 XHTML

A variation of TEXT but in HTML format, containing the bare text and images ("semantic" output):

```
<div>
<p><img width="195" height="86" src="
/9j/4AAQSkZJRgABAQEAYABgAAD/4Q (... omitted image data ...)"/></p>
<p><b>PyMuPDF Documentation</b></p>
<p><b><i>Release 1.12.0</i></b></p>
<p><b>Jorj X. McKie</b></p>
<p><b>Dec 13, 2017</b></p>
</div>
```

10.8 Further Remarks

1. We have modified MuPDF's **plain text** extraction: The original prints out every line followed by a newline character. This leads to a rather ragged, space-wasting look. So we have combined all lines of a text block into one, separating lines by space characters (but only if a line does not end with "-"). We also do not add extra newline characters at the end of blocks.
2. The 5 extraction methods each have a default behavior concerning images: "TEXT" and "XML" do not extract images, while the other three do. On occasion it may make sense to switch off images for "HTML", "XHTML" or "JSON", too. See chapter *Working together: DisplayList and TextPage* on how to achieve this. Use an argument of `3` when you create the *TextPage*.

3. Apart from the 5 standard ones, we offer additional extraction methods *Page.getTextBlocks()* and *Page.getTextWords()*. They return lists of a page's text blocks, resp. words. Each list item contains text accompanied by its rectangle ("bbox", location on the page). This should help to resolve extraction issues around multi-column or boxed text.
4. If you need even more detailed positioning information, you can use XML extraction.

10.9 Performance

The text extraction methods differ significantly: in terms of information they supply (see above), and in terms of resource requirements. More information of course means that more processing is required and a higher data volume is generated.

To begin with, all methods are **very** fast in relation to what is out there on the market. In terms of processing speed, we couldn't find a faster (free) tool. Even the most detailed method, XML, processes all 1'310 pages of the *Adobe PDF Reference 1.7* in less than 8 seconds.

Relative to each other, **"XML"** is about 2 times slower than **"TEXT"**, the others range between them. E.g. **"JSON", "HTML", "XHTML"** need about 20% more time than **"TEXT"** (heavily depending on the size of images contained in the document), whereas *Page.getTextBlocks()* and *Page.getTextWords()* are only 1% resp. 3% slower.

Look into the previous chapter **Appendix 1** for more performance information.

CHAPTER

ELEVEN

APPENDIX 3: CONSIDERATIONS ON EMBEDDED FILES

This chapter provides some background on embedded files support in PyMuPDF.

11.1 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 files a PDF may reference or even 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.

11.2 MuPDF Support

MuPDF v1.11 added initial support for embedded files and collections (also called *portfolios*).

The library contains functions to add files to the `EmbeddedFiles` name tree and display some information of its entries.

Also supported is a full set of functions to maintain collections (advanced metadata maintenance) and their relation to embedded files.

11.3 PyMuPDF Support

Starting with PyMuPDF v1.11.0 we fully reflect MuPDF's support for embedded files and partly go beyond that scope:

- We can add, extract **and** delete embedded files.
- We can display **and** change some meta information (outside collections). Informations available for display are **name**, **filename**, **description**, **length** and compressed **size**. Of these properties, *filename* and *description* can also be changed, after a file has been embedded.

Support of the *collections* feature has been postponed to a later version. We will probably include this ever only on user request.

CHAPTER

TWELVE

APPENDIX 4: ASSORTED TECHNICAL INFORMATION

12.1 PDF Base 14 Fonts

The following 14 builtin font names must be supported by every PDF aplication. They are available as the Python list `fitz.Base14_Fonts`:

- Courier
- Courier-Oblique
- Courier-Bold
- Courier-BoldOblique
- Helvetica
- Helvetica-Oblique
- Helvetica-Bold
- Helvetica-BoldOblique
- Times-Roman
- Times-Bold
- Times-Italic
- Times-BoldItalic
- Symbol
- ZapfDingbats

12.2 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: http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf.

12.3 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.

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 between a page and its links and 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.

CHAPTER

THIRTEEN

CHANGE LOGS

13.1 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 remains precise across zooming. Both involved documents must be PDF.
- New method *Page.getSVGimage()* creates an SVG image from the page. 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), images contained in the *TextPage* 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.

13.2 Changes in Version 1.12.0

This version is based on and requires MuPDF v1.12. 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 have not 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 text pages, 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.

13.3 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.

13.4 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, rectangle-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)`.

13.5 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.

13.6 Changes in Version 1.10.0

13.6.1 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 alltogether. 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.

13.6.2 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.

13.7 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.

13.8 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 `str` (Python 2), `bytes` (Python 3) or `bytearray` (Python 2 and 3). 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 `str` (Python 2) or `bytes` (Python 3), not only `bytearray`.

13.9 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.

CHAPTER

FOURTEEN

ERROR MESSAGES

This a list of exception messages raised by PyMuPDF together with an explanation and possible solution.

In addition, the underlying C library MuPDF also raises exceptions on the Python level. We have included a few of those as well and may extend this in future.

annot has no /AP

- Bad specification - no changes possible for this annotation.

arg 1 not bytes or bytearray

- Specify parameter as type `bytes` or `bytearray`.

bad PDF: Contents is no stream object

- The `/Contents` object(s) of a page must be streams. Repair PDF.

bad PDF: file has no stream

- An embedded / attached file is not a stream. Repair PDF.

buffer too large to deflate

- Internal error - report an issue.

cannot deflate buffer

- Internal error - report an issue.

cannot open <path>: No such file or directory

- Specify a valid file name / path.

cannot recognize archive

- Trying to open an invalid CBZ document.

cannot recognize zip archive

- Trying to open an invalid XPS document.

color components must be in range 0 to 1

- Color components must be floats in interval [0, 1].

could not create UTF16 for '<name>'

- Internal error - report an issue.

could not get string of '<name>'

- Internal error - report an issue.

could not get UTF16 string of '<name>'

- Internal error - report an issue.

could not load root object

- Root object of PDF not found. Repair PDF.

encrypted file - save to new

- Trying incremental save for a decrypted file. Use `doc.save()` to a new file.

exactly one of filename, pixmap must be given

- You either specified both parameters or none.

expected a sequence

- Parameter type must be `list`, `tuple`, etc.

filename must be a string

- Specify a valid file path / name.

filename must be string or None

- Specify a valid file path / name or omit parameter.

filename must end with '.png'

- `writePNG()` requires file extension `.png`.

filetype missing with stream specified

- Document open from memory needs its type as a string.

fontname must be supplied

- A new font file requires some (arbitrary) **new** reference name.

found code point nnn: increase charlimit

- Trying to get a glyph width beyond the current table size limit.

incremental excludes garbage

- Garbage collection cannot occur during incremental saves.

incremental excludes linear

- Linearization cannot occur during incremental saves.

incremental save needs original file

- Incremental save is only possible to the original file.

info not a dict

- Specify correct Python parameter type.

invalid font - FontDescriptor missing

- Specify correct XREF to read font.

invalid font descriptor subtype

- Bad font description in PDF. Repair file.

unhandled font type / unhandled font type '<type>'

- MuPDF does not yet handle this font type. Requesting method cannot be used, unfortunately. Report an issue.

invalid key in info dict

- Dictionary key misspelled.

invalid page range

- Page numbers must be in range `[0, pageCount - 1]`.

invalid stream

- Stream object updates need type `bytes` or `bytearray`.

len(samples) invalid

- Length of samples must equal `width * height * n` (where `n` is the number of components per pixel).

line endpoints must be within page rect

- The `Page.rect` must contain the points.

name already exists

- The name is in use by some other embedded file.

name not valid

- Specify a name of non-zero length.

need 3 color components

- Only RGB colors are supported, which need three components.

no embedded files

- PDF has no embedded files.

no objects found

- Trying to open an invalid PDF, FB2, or EPUB document.

not a file attachment annot

- Accessed an annotation with the wrong type.

not a PDF

- Using some method or attribute only valid for PDF document type.

nothing to change

- No data supplied for embedded file metadata change.

operation illegal for closed doc

- Trying to use methods / properties after close of document.

orphaned object: parent is None

- Accessing an object whose parent no longer exists (e.g. an annotation of an unavailable page).

page number out of range

- Page numbers must always be `< pageCount`, but also non-negative for some methods.

page numbers must be integers

- Specify valid page numbers (`select()` method).

rect must be contained in page rect

- Image insertion requires a target rectangle contained in `page.rect`.

rect must be finite and not empty

- Top-left corner must be "northeast" of bottom-right one, and rectangle area must be positive.

repaired file - save to new

- Trying incremental save for file repaired during open. Use `doc.save()` to a new file.

save to original requires incremental

- Using original filename in `doc.save()` without also specifying option `incremental`. Consider using `doc.saveIncr()`.

sequence length must be <n>

- Creating Point, Rect, Irect, Matrix with wrong length sequences.

some text is needed

- Specify text with a positive length.

source and target too close

- Target number of moved page `pno` must be `> pno` or `< pno - 1`.

source must not equal target PDF

- Method `doc.insertPDF()` requires two distinct document objects (which may point to the same file, however).

source not a PDF

- Method `doc.insertPDF()` only works with PDF documents.

source page out of range

- Specify a valid page number.

target not a PDF

- Method `doc.insertPDF()` only works with PDF documents.

text position outside page height range

- If text starts at *Point* `point`, `fontsize <= point.y <= (page height - fontsize * 1.2)` must be true.

type(ap) invalid

- Internal error - report an issue.

type(imagedata) invalid

- Use type `bytearray`.

type(samples) invalid

- Use type `bytes` or `bytearray`.

unknown PDF Base 14 font

- Use a valid PDF standard font name.

xref entry is not an image

- Trying to create a pixmap from a non-image PDF object.

xref invalid

- Internal error - report an issue.

xref is not a stream

- Trying to access the stream part of a non-stream object.

xref out of range

- PDF xref numbers must be `1 <= xref <= doc._getXrefLength()`.

INDEX