



PyMuPDF Documentation

Release 1.18.5

Jorj X. McKie

Dec 17, 2020

1	Introduction	1
1.1	Note on the Name <i>fitz</i>	2
1.2	License	2
1.3	Covered Version	2
2	Installation	3
2.1	Option 1: Install from Sources	3
2.1.1	Step 1: Download PyMuPDF	3
2.1.2	Step 2: Download and Generate MuPDF	3
2.1.3	Step 3: Build / Setup PyMuPDF	4
2.2	Option 2: Install from Binaries	4
3	Tutorial	7
3.1	Importing the Bindings	7
3.2	Opening a Document	7
3.3	Some Document Methods and Attributes	8
3.4	Accessing Meta Data	8
3.5	Working with Outlines	8
3.6	Working with Pages	9
3.6.1	Inspecting the Links, Annotations or Form Fields of a Page	9
3.6.2	Rendering a Page	10
3.6.3	Saving the Page Image in a File	10
3.6.4	Displaying the Image in GUIs	10
3.6.4.1	wxPython	10
3.6.4.2	Tkinter	11
3.6.4.3	PyQt4, PyQt5, PySide	11
3.6.5	Extracting Text and Images	11
3.6.6	Searching for Text	12
3.7	PDF Maintenance	12
3.7.1	Modifying, Creating, Re-arranging and Deleting Pages	12
3.7.2	Joining and Splitting PDF Documents	13
3.7.3	Embedding Data	13
3.7.4	Saving	13
3.8	Closing	14
3.9	Further Reading	14
4	Collection of Recipes	15

4.1	Images	15
4.1.1	How to Make Images from Document Pages	15
4.1.2	How to Increase Image Resolution	16
4.1.3	How to Create Partial Pixmaps (Clips)	16
4.1.4	How to Create or Suppress Annotation Images	17
4.1.5	How to Extract Images: Non-PDF Documents	17
4.1.6	How to Extract Images: PDF Documents	17
4.1.7	How to Handle Stencil Masks	19
4.1.8	How to Make one PDF of all your Pictures (or Files)	20
4.1.9	How to Create Vector Images	22
4.1.10	How to Convert Images	23
4.1.11	How to Use Pixmaps: Gluing Images	24
4.1.12	How to Use Pixmaps: Making a Fractal	25
4.1.13	How to Interface with NumPy	27
4.1.14	How to Add Images to a PDF Page	27
4.2	Text	28
4.2.1	How to Extract all Document Text	28
4.2.2	How to Extract Text from within a Rectangle	29
4.2.3	How to Extract Text in Natural Reading Order	29
4.2.4	How to Extract Tables from Documents	31
4.2.5	How to Search for and Mark Text	31
4.2.6	How to Analyze Font Characteristics	33
4.2.7	How to Insert Text	34
4.2.7.1	How to Write Text Lines	35
4.2.7.2	How to Fill a Text Box	36
4.2.7.3	How to Use Non-Standard Encoding	37
4.3	Annotations	38
4.3.1	How to Add and Modify Annotations	39
4.3.2	How to Mark Text	43
4.3.3	How to Use FreeText	44
4.3.4	Using Buttons and JavaScript	45
4.3.5	How to Use Ink Annotations	47
4.4	Drawing and Graphics	48
4.5	Extracting Drawings	50
4.6	Multiprocessing	52
4.7	General	57
4.7.1	How to Open with a Wrong File Extension	57
4.7.2	How to Embed or Attach Files	57
4.7.3	How to Delete and Re-Arrange Pages	58
4.7.4	How to Join PDFs	59
4.7.5	How to Add Pages	59
4.7.6	How To Dynamically Clean Up Corrupt PDFs	60
4.7.7	How to Split Single Pages	61
4.7.8	How to Combine Single Pages	63
4.7.9	How to Convert Any Document to PDF	64
4.7.10	How to Deal with Messages Issued by MuPDF	65
4.7.11	How to Deal with PDF Encryption	66
4.8	Common Issues and their Solutions	68
4.8.1	Changing Annotations: Unexpected Behaviour	68
4.8.1.1	Problem	68
4.8.1.2	Cause	68
4.8.1.3	Solutions	69
4.8.2	Misplaced Item Insertions on PDF Pages	69
4.8.2.1	Problem	69

4.8.2.2	Cause	69
4.8.2.3	Solutions	70
4.9	Low-Level Interfaces	71
4.9.1	How to Iterate through the <code>xref</code> Table	71
4.9.2	How to Handle Object Streams	72
4.9.3	How to Handle Page Contents	72
4.9.4	How to Access the PDF Catalog	73
4.9.5	How to Access the PDF File Trailer	74
4.9.6	How to Access XML Metadata	74
5	Using <i>fitz</i> as a Module	77
5.1	Invocation	77
5.2	Cleaning and Copying	78
5.3	Extracting Fonts and Images	79
5.4	Joining PDF Documents	79
5.5	Low Level Information	80
5.6	Embedded Files Commands	81
5.6.1	Information	81
5.6.2	Extraction	82
5.6.3	Deletion	83
5.6.4	Insertion	83
5.6.5	Updates	83
5.6.6	Copying	84
6	Classes	85
6.1	Annot	85
6.1.1	Annotation Icons in MuPDF	95
6.1.2	Example	96
6.2	Colorspace	97
6.3	DisplayList	98
6.4	Document	99
6.4.1	<code>setMetadata()</code> Example	128
6.4.2	<code>setToC()</code> Demonstration	128
6.4.3	<code>insertPDF()</code> Examples	129
6.4.4	Other Examples	129
6.5	Font	130
6.6	Identity	135
6.7	IRect	136
6.8	Link	139
6.9	linkDest	141
6.10	Matrix	142
6.10.1	Examples	146
6.10.2	Shifting	146
6.10.3	Flipping	147
6.10.4	Shearing	148
6.10.5	Rotating	149
6.11	Outline	150
6.12	Page	152
6.12.1	Modifying Pages	152
6.12.2	Description of <code>getLinks()</code> Entries	177
6.12.3	Notes on Supporting Links	178
6.12.3.1	Reading (pertains to method <code>getLinks()</code> and the <code>firstLink</code> property chain)	178
6.12.3.2	Writing	178
6.12.4	Homologous Methods of Document and Page	178

6.13	Pixmap	179
6.13.1	Supported Input Image Formats	187
6.13.2	Supported Output Image Formats	187
6.14	Point	188
6.15	Quad	190
6.15.1	Remark	193
6.16	Rect	193
6.17	Shape	198
6.17.1	Usage	209
6.17.2	Examples	209
6.17.3	Common Parameters	210
6.18	TextPage	213
6.18.1	Dictionary Structure of <code>extractDICT()</code> and <code>extractRAWDICT()</code>	217
6.18.1.1	Page Dictionary	217
6.18.1.2	Block Dictionaries	217
6.18.1.3	Line Dictionary	218
6.18.1.4	Span Dictionary	219
6.18.1.5	Character Dictionary for <code>extractRAWDICT()</code>	220
6.19	TextWriter	220
6.20	Tools	224
6.20.1	Example Session	228
6.21	Widget	229
6.21.1	Standard Fonts for Widgets	231
6.21.2	Supported Widget Types	232
7	Operator Algebra for Geometry Objects	233
7.1	General Remarks	233
7.2	Unary Operations	234
7.3	Binary Operations	234
7.4	Some Examples	235
7.4.1	Manipulation with numbers	235
7.4.2	Manipulation with “like” Objects	235
8	Low Level Functions and Classes	239
8.1	Functions	239
8.2	Device	254
8.3	Working together: <code>DisplayList</code> and <code>TextPage</code>	255
8.3.1	Create a <code>DisplayList</code>	255
8.3.2	Generate Pixmap	255
8.3.3	Perform Text Search	255
8.3.4	Extract Text	256
8.3.5	Further Performance improvements	256
8.3.5.1	Pixmap	256
8.3.5.2	TextPage	256
9	Glossary	257
10	Constants and Enumerations	261
10.1	Constants	261
10.2	Document Permissions	262
10.3	PDF encryption method codes	262
10.4	Font File Extensions	263
10.5	Text Alignment	263
10.6	Preserve Text Flags	263
10.7	Link Destination Kinds	264

10.8	Link Destination Flags	264
10.9	Annotation Related Constants	265
10.9.1	Annotation Types	265
10.9.2	Annotation Flag Bits	266
10.9.3	Annotation Line Ending Styles	266
10.10	Widget Constants	266
10.10.1	Widget Types (<i>field_type</i>)	266
10.10.2	Text Widget Subtypes (<i>text_format</i>)	266
10.10.3	Widget flags (<i>field_flags</i>)	267
10.11	PDF Standard Blend Modes	267
10.12	Stamp Annotation Icons	268
11	Color Database	269
11.1	Function <i>getColor()</i>	269
11.2	Printing the Color Database	270
12	Appendix 1: Performance	271
12.1	Part 1: Parsing	272
12.2	Part 2: Text Extraction	274
12.3	Part 3: Image Rendering	275
13	Appendix 2: Details on Text Extraction	279
13.1	General structure of a <i>TextPage</i>	279
13.2	Plain Text	280
13.3	BLOCKS	280
13.4	WORDS	280
13.5	HTML	281
13.6	Controlling Quality of HTML Output	281
13.7	DICT (or JSON)	282
13.8	RAWDICT	283
13.9	XML	283
13.10	XHTML	284
13.11	Text Extraction Flags Defaults	284
13.12	Performance	285
14	Appendix 3: Considerations on Embedded Files	287
14.1	General	287
14.2	MuPDF Support	287
14.3	PyMuPDF Support	288
15	Appendix 4: Assorted Technical Information	289
15.1	PDF Base 14 Fonts	289
15.2	Adobe PDF References	290
15.3	Using Python Sequences as Arguments in PyMuPDF	290
15.4	Ensuring Consistency of Important Objects in PyMuPDF	291
15.5	Design of Method <i>Page.showPDFpage()</i>	293
15.5.1	Purpose and Capabilities	293
15.5.2	Technical Implementation	293
15.6	Redirecting Error and Warning Messages	294
16	Change Logs	295
16.1	Changes in Version 1.18.5	295
16.2	Changes in Version 1.18.4	296
16.3	Changes in Version 1.18.3	296
16.4	Changes in Version 1.18.2	297

16.5	Changes in Version 1.18.1	297
16.6	Changes in Version 1.18.0	297
16.7	Changes in Version 1.17.7	298
16.8	Changes in Version 1.17.6	299
16.9	Changes in Version 1.17.5	299
16.10	Changes in Version 1.17.4	299
16.11	Changes in Version 1.17.3	300
16.12	Changes in Version 1.17.2	300
16.13	Changes in Version 1.17.1	300
16.14	Changes in Version 1.17.0	300
16.15	Changes in Version 1.16.18	301
16.16	Changes in Version 1.16.17	301
16.17	Changes in Version 1.16.16	301
16.18	Changes in Version 1.16.14	302
16.19	Changes in Version 1.16.13	302
16.20	Changes in Version 1.16.12	302
16.21	Changes in Version 1.16.11	302
16.22	Changes in Version 1.16.10	303
16.23	Changes in Version 1.16.9	303
16.24	Changes in Version 1.16.8	303
16.25	Changes in Version 1.16.7	303
16.26	Changes in Version 1.16.6	304
16.27	Changes in Version 1.16.5	304
16.28	Changes in Version 1.16.4	304
16.29	Changes in Version 1.16.3	304
16.30	Changes in Version 1.16.2	305
16.31	Changes in Version 1.16.1	305
16.32	Changes in Version 1.16.0	305
16.33	No version published for MuPDF v1.15.0	306
16.34	Changes in Version 1.14.20 / 1.14.21	306
16.35	Changes in Version 1.14.19	307
16.36	Changes in Version 1.14.17	307
16.37	Changes in Version 1.14.16	307
16.38	Changes in Version 1.14.15	307
16.39	Changes in Version 1.14.14	307
16.40	Changes in Version 1.14.13	308
16.41	Changes in Version 1.14.12	308
16.42	Changes in Version 1.14.11	308
16.43	Changes in Version 1.14.10	308
16.44	Changes in Version 1.14.9	308
16.45	Changes in Version 1.14.8	309
16.46	Changes in Version 1.14.7	309
16.47	Changes in Version 1.14.5	309
16.48	Changes in Version 1.14.4	310
16.49	Changes in Version 1.14.3	310
16.50	Changes in Version 1.14.1	310
16.51	Changes in Version 1.14.0	310
16.52	Changes in Version 1.13.19	311
16.53	Changes in Version 1.13.18	311
16.54	Changes in Version 1.13.17	311
16.55	Changes in Version 1.13.16	312
16.56	Changes in Version 1.13.15	312
16.57	Changes in Version 1.13.14	312
16.58	Changes in Version 1.13.13	312

16.59 Changes in Version 1.13.12	313
16.60 Changes in Version 1.13.11	313
16.61 Changes in Version 1.13.7	313
16.62 Changes in Version 1.13.6	314
16.63 Changes in Version 1.13.5	314
16.64 Changes in Version 1.13.4	314
16.65 Changes in Version 1.13.3	314
16.66 Changes in Version 1.13.2	314
16.67 Changes in Version 1.13.1	315
16.68 Changes in Version 1.13.0	315
16.69 Changes in Version 1.12.4	315
16.70 Changes in Version 1.12.3	316
16.71 Changes in Version 1.12.2	316
16.72 Changes in Version 1.12.1	316
16.73 Changes in Version 1.12.0	316
16.74 Changes in Version 1.11.2	317
16.75 Changes in Version 1.11.1	317
16.76 Changes in Version 1.11.0	318
16.77 Changes in Version 1.10.0	318
16.77.1 MuPDF v1.10 Impact	318
16.77.2 Other Changes compared to Version 1.9.3	319
16.78 Changes in Version 1.9.3	320
16.79 Changes in Version 1.9.2	320
16.80 Changes in Version 1.9.1	321

Index



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 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.

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

PyMuPDF is hosted on [GitHub](#). We also are registered on [PyPI](#).

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

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

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

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

For installation details check out the respective chapter.

There exist several [demo](#) 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 v1.18.5 features as of **2020-12-17 04:49:36**.

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

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

2.1 Option 1: Install from Sources

This is a three-step process.

2.1.1 Step 1: Download PyMuPDF

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

2.1.2 Step 2: Download and Generate MuPDF

Download *mupdf-x.xx.x-source.tar.gz* from [Mupdf](#) and unzip / decompress it. Make sure to download the (sub-) version for which PyMuPDF has stated its compatibility.

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

Applying any Changes and Hot Fixes to MuPDF Sources

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

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

- *_config.h* – **(Optional)** PyMuPDF’s configuration to control the binary file size. Copy-rename it to */include/mupdf/fitz/config.h*. This reduces the size of the PyMuPDF binary extension module to around 11 MB. If omitting this change, that size will be over 30 MB – without impacting functionality.

Generate MuPDF

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

PyMuPDF's [homepage](#) contains additional details and hints.

2.1.3 Step 3: Build / Setup PyMuPDF

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

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

Now perform a `python setup.py install`.

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

2.2 Option 2: Install from Binaries

You can install PyMuPDF from Python wheels. The wheels are *self-contained*, i.e. you will **not need any other software** nor download / install MuPDF to run PyMuPDF scripts. This installation option is available for all MS Windows and the most **popular 64-bit** Mac OSX and Linux platforms for Python versions 3.6 through 3.9. Windows binaries are provided for Python **32-bit and 64-bit** versions.

Note: For the time being, wheels for Python versions 2.7 and 3.5 are generated as well, but not uploaded to PyPI until explicitly requested via an issue. Starting year 2021, support for these wheel versions will be dropped entirely.

Overview of wheel names (PyMuPDF version is x.xx.xx):

```
PyMuPDF-x.xx.xx-cp36-cp36m-macosx_10_9_x86_64.whl
PyMuPDF-x.xx.xx-cp36-cp36m-manylinux2010_x86_64.whl
PyMuPDF-x.xx.xx-cp36-cp36m-win32.whl
PyMuPDF-x.xx.xx-cp36-cp36m-win_amd64.whl
PyMuPDF-x.xx.xx-cp37-cp37m-macosx_10_9_x86_64.whl
PyMuPDF-x.xx.xx-cp37-cp37m-manylinux2010_x86_64.whl
PyMuPDF-x.xx.xx-cp37-cp37m-win32.whl
PyMuPDF-x.xx.xx-cp37-cp37m-win_amd64.whl
PyMuPDF-x.xx.xx-cp38-cp38-macosx_10_9_x86_64.whl
PyMuPDF-x.xx.xx-cp38-cp38-manylinux2010_x86_64.whl
PyMuPDF-x.xx.xx-cp38-cp38-win32.whl
PyMuPDF-x.xx.xx-cp38-cp38-win_amd64.whl
PyMuPDF-x.xx.xx-cp39-cp39-macosx_10_9_x86_64.whl
PyMuPDF-x.xx.xx-cp39-cp39-manylinux2010_x86_64.whl
PyMuPDF-x.xx.xx-cp39-cp39-win32.whl
PyMuPDF-x.xx.xx-cp39-cp39-win_amd64.whl
```

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

If you unexpectedly run into problems installing the wheel for your system, please make sure you have updated your PIP to the current version.

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 PyMuPDF¹. Nevertheless, for the sake of brevity we will only talk about PDF files. At places where indeed only PDF files are supported, this will be mentioned explicitly.

3.1 Importing the Bindings

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

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

3.2 Opening a Document

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

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

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

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

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

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

3.3 Some Document Methods and Attributes

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

3.4 Accessing Meta Data

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

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

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

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

3.5 Working with Outlines

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

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

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

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

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

3.6 Working with Pages

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

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

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

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

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

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

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

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

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

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

3.6.1 Inspecting the Links, Annotations or Form Fields of a Page

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

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

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

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

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

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

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

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

3.6.2 Rendering a Page

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

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

pix is a *Pixmap* object which (in this case) contains an **RGB** image of the page, ready to be used for many purposes. Method *Page.getPixmap()* offers lots of variations for controlling the image: resolution, colorspace (e.g. to produce a grayscale image or an image with a subtractive color scheme), transparency, rotation, mirroring, shifting, shearing, etc. For example: to create an **RGBA** image (i.e. containing an alpha channel), specify *pix = page.getPixmap(alpha=True)*.

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

Note: You can also create a **vector** image of a page by using *Page.getSVGimage()*. Refer to this [Wiki](#) for details.

3.6.3 Saving the Page Image in a File

We can simply store the image in a PNG file:

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

3.6.4 Displaying the Image in GUIs

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

3.6.4.1 wxPython

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

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

3.6.4.2 Tkinter

Please also see section 3.19 of the [Pillow documentation](#):

```
from PIL import Image, ImageTk

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

The following avoids using **Pillow**:

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

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

3.6.4.3 PyQt4, PyQt5, PySide

Please also see section 3.16 of the [Pillow documentation](#):

```
from PIL import Image, ImageQt

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

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

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

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

3.6.5 Extracting Text and Images

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

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

Use one of the following strings for *opt* to obtain different formats²:

- “*text*”: (default) plain text with line breaks. No formatting, no text position details, no images.
- “*blocks*”: generate a list of text blocks (= paragraphs).
- “*words*”: generate a list of words (strings not containing spaces).

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

- “*html*”: creates a full visual version of the page including any images. This can be displayed with your internet browser.
- “*dict*” / “*json*”: same information level as HTML, but provided as a Python dictionary or resp. JSON string. See `TextPage.extractDICT()` resp. `TextPage.extractJSON()` for details of its structure.
- “*rawdict*” / “*rawjson*”: a super-set of `TextPage.extractDICT()`. It additionally provides character detail information like XML. See `TextPage.extractRAWDICT()` for details of its structure.
- “*xhtml*”: text information level as the TEXT version but includes images. Can also be displayed by internet browsers.
- “*xml*”: contains no images, but full position and font information down to each single text character. Use an XML module to interpret.

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

3.6.6 Searching for Text

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

```
areas = page.searchFor("mupdf")
```

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

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

3.7 PDF Maintenance

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

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

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

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

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

3.7.1 Modifying, Creating, Re-arranging and Deleting Pages

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

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

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

`Document.select()` shrinks a PDF down to selected pages. Parameter is a sequence³ of the page numbers that you want to keep. These integers must all be in range $0 \leq i < \text{pageCount}$. When executed, all pages **missing** in this list will be deleted. Remaining pages will occur **in the sequence and as many times (!) as you specify them**.

So you can easily create new PDFs with

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

... whatever you can think of.

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

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

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

3.7.2 Joining and Splitting PDF Documents

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

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

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

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

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

3.7.3 Embedding Data

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

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

3.7.4 Saving

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

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

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

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

Save Option	mutool	Effect
<code>garbage=1</code>	<code>g</code>	garbage collect unused objects
<code>garbage=2</code>	<code>gg</code>	in addition to 1, compact <i>xref</i> tables
<code>garbage=3</code>	<code>ggg</code>	in addition to 2, merge duplicate objects
<code>garbage=4</code>	<code>gggg</code>	in addition to 3, merge duplicate stream content
<code>clean=True</code>	<code>cs</code>	clean and sanitize content streams
<code>deflate=True</code>	<code>z</code>	deflate uncompressed streams
<code>deflate_images=True</code>	<code>i</code>	deflate image streams
<code>deflate_fonts=True</code>	<code>f</code>	deflate fontfile streams
<code>ascii=True</code>	<code>a</code>	convert binary data to ASCII format
<code>linear=True</code>	<code>l</code>	create a linearized version
<code>expand=True</code>	<code>d</code>	decompress all streams

Note: For an explanation of terms like *object*, *stream*, *xref* consult the *Glossary* chapter.

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

3.8 Closing

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

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

3.9 Further Reading

Also have a look at PyMuPDF's [Wiki](#) pages. Especially those named in the sidebar under title “**Recipes**” cover over 15 topics written in “How-To” style.

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

Collection of Recipes

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

4.1 Images

4.1.1 How to Make Images from Document Pages

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

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

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

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

The script directory will now contain PNG image files named *page-0.png*, *page-1.png*, etc. Pictures have the dimension of their pages, e.g. 595 x 842 pixels for an A4 portrait sized page. They will have a resolution of 72 dpi in x and y dimension and have no transparency. You can change all that – for how to do this, read the next sections.

4.1.2 How to Increase Image Resolution

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

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

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

In the following, we apply a zoom factor of 2 to each dimension, which will generate an image with a four times better resolution for us (and also about 4 times the size):

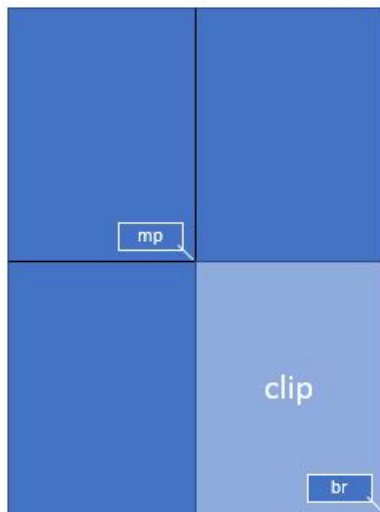
```
zoom_x = 2.0 # horizontal zoom
zomm_y = 2.0 # vertical zoom
mat = fitz.Matrix(zoom_x, zomm_y) # zoom factor 2 in each dimension
pix = page.getPixmap(matrix = mat) # use 'mat' instead of the identity matrix
```

4.1.3 How to Create Partial Pixmap (Clips)

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

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

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



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

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

4.1.4 How to Create or Suppress Annotation Images

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

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

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

4.1.5 How to Extract Images: Non-PDF Documents

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

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

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

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

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

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

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

```
{"type": 1, "bbox": (x0, y0, x1, y1), "width": w, "height": h, "ext": "png", "image": "\u2192b"...}
```

4.1.6 How to Extract Images: PDF Documents

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

1. **Create** a `Pixmap` of the image with instruction `pix = fitz.Pixmap(doc, xref)`. This method is **very** fast (single digit micro-seconds). The pixmap’s properties (width, height, ...) will reflect the ones of the image. In this case there is no way to tell which image format the embedded original has.

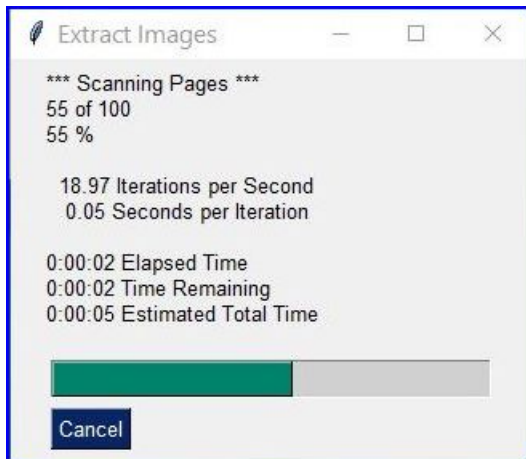
2. **Extract** the image with `img = doc.extractImage(xref)`. This is a dictionary containing the binary image data as `img["image"]`. A number of meta data are also provided – mostly the same as you would find in the pixmap of the image. The major difference is string `img["ext"]`, which specifies the image format: apart from “png”, strings like “jpeg”, “bmp”, “tiff”, etc. can also occur. Use this string as the file extension if you want to store to disk. The execution speed of this method should be compared to the combined speed of the statements `pix = fitz.Pixmap(doc, xref); pix.getPNGData()`. If the embedded image is in PNG format, the speed of `Document.extractImage()` is about the same (and the binary image data are identical). Otherwise, this method is **thousands of times faster**, and the **image data is much smaller**.

The question remains: “**How do I know those ‘xref’ numbers of images?**”. There are two answers to this:

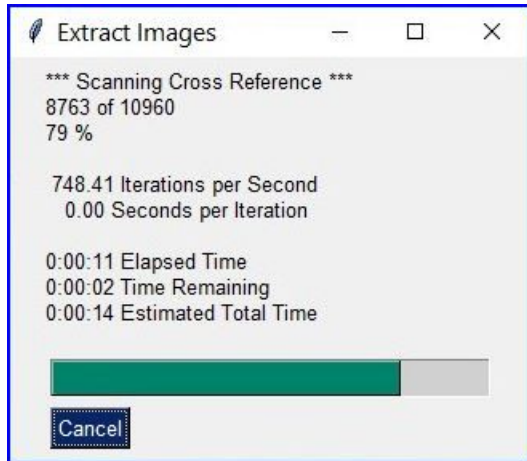
- a. “**Inspect the page objects:**” Loop through the items of `Page.getImageList()`. It is a list of list, and its items look like `[xref, smask, ...]`, containing the `xref` of an image. This `xref` can then be used with one of the above methods. Use this method for **valid (undamaged)** documents. Be wary however, that the same image may be referenced multiple times (by different pages), so you might want to provide a mechanism avoiding multiple extracts.
- b. “**No need to know:**” Loop through the list of **all xrefs** of the document and perform a `Document.extractImage()` for each one. If the returned dictionary is empty, then continue – this `xref` is no image. Use this method if the PDF is **damaged (unusable pages)**. Note that a PDF often contains “pseudo-images” (“stencil masks”) with the special purpose of defining the transparency of some other image. You may want to provide logic to exclude those from extraction. Also have a look at the next section.

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

`extract-imga.py` extracts images page by page:



and `extract-imgb.py` extracts images by xref table:



4.1.7 How to Handle Stencil Masks

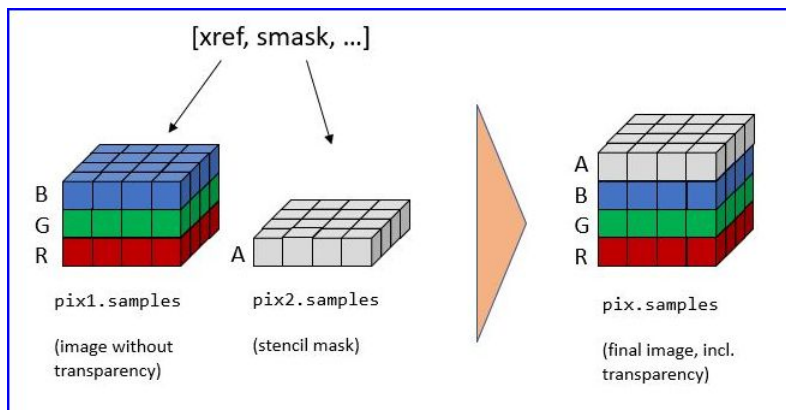
Some images in PDFs are accompanied by **stencil masks**. In their simplest form stencil masks represent alpha (transparency) bytes stored as separate images. In order to reconstruct the original of an image, which has a stencil mask, it must be “enriched” with transparency bytes taken from its stencil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

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

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

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

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



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

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

The scripts `extract-imga.py`, and `extract-imgb.py` above also contain this logic.

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

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

Method 1: Inserting Images as Pages

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

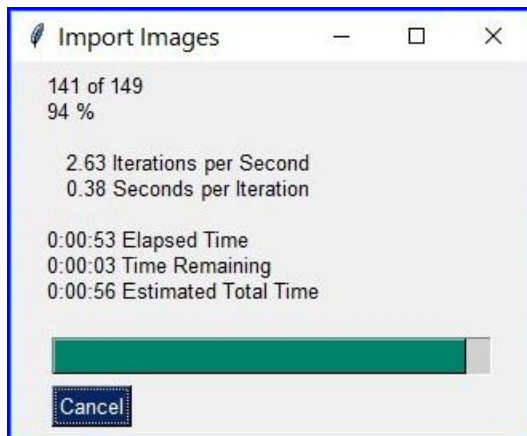
```
import os, fitz
import PySimpleGUI as psq # for showing a progress bar
doc = fitz.open() # PDF with the pictures
imgdir = "D:/2012_10_05" # where the pics are
imglist = os.listdir(imgdir) # list of them
imgcount = len(imglist) # pic count

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

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

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

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



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

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

Method 2: Embedding Files

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

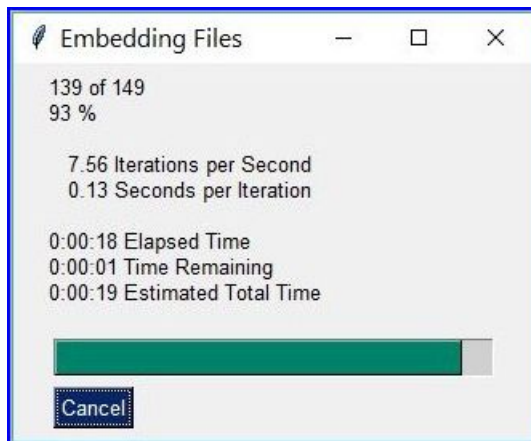
```
import os, fitz
import PySimpleGUI as psq # for showing progress bar
doc = fitz.open() # PDF with the pictures
imgdir = "D:/2012_10_05" # where my files are

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

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

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

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

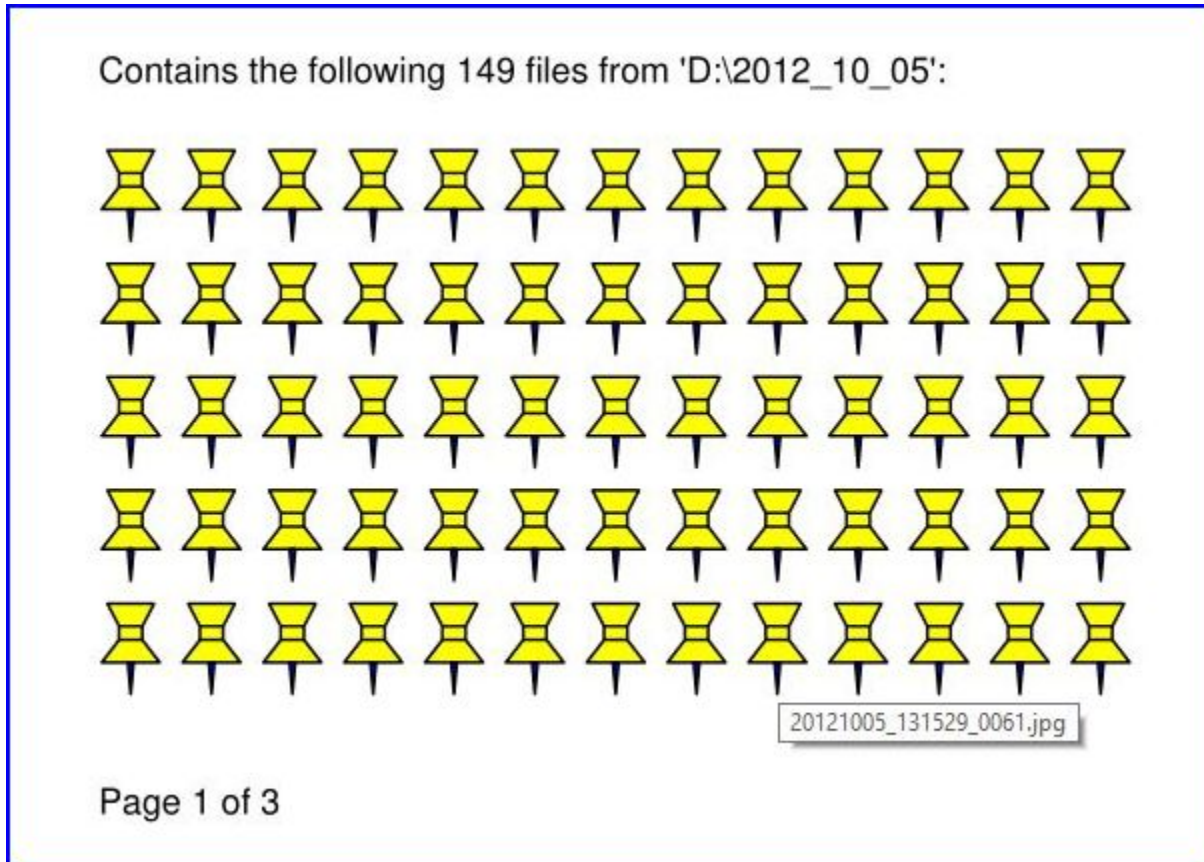


This is by far the fastest method, and it also produces the smallest possible output file size. The above pictures needed 20 seconds on my machine and yielded a PDF size of 510 MB. Look [here](#) for a more complete source code: it offers a directory selection dialog and skips non-file entries.

Method 3: Attaching Files

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

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



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

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

4.1.9 How to Create Vector Images

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

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

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

4.1.10 How to Convert Images

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

Notwithstanding that interfacing with Pillow is almost trivial.

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

The general scheme is just the following two lines:

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

Remarks

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

Note: Convert JPEG to Photoshop:

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

Note: Save to JPEG using PIL/Pillow:

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

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

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

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

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

4.1.11 How to Use Pixmaps: Gluing Images

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

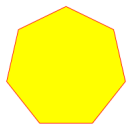
```
import fitz
src = fitz.Pixmap("img-7edges.png") # create pixmap from a picture
col = 3 # tiles per row
lin = 4 # tiles per column
tar_w = src.width * col # width of target
tar_h = src.height * lin # height of target

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

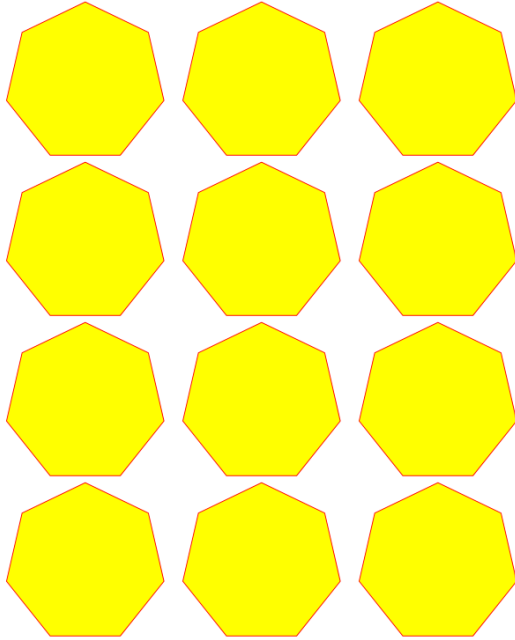
# now fill target with the tiles
for i in range(col):
    for j in range(lin):
        src.setOrigin(src.width * i, src.height * j)
        tar_pix.copyPixmap(src, src.irect) # copy input to new loc

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

This is the input picture:



Here is the output:



4.1.12 How to Use Pixmaps: Making a Fractal

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

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

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

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

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

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

# alternatively, define a 'fill' pixmap for the punch holes
# this could be anything, e.g. some photo image ...
fill = fitz.Pixmap(fitz.csRGB, ir, False) # same size as 'pm'
fill.setRect(fill.irect, (0, 255, 255)) # put some color in

def punch(x, y, step):
    """Recursively "punch a hole" in the central square of a pixmap.
```

(continues on next page)

(continued from previous page)

```

Arguments are top-left coords and the step width.

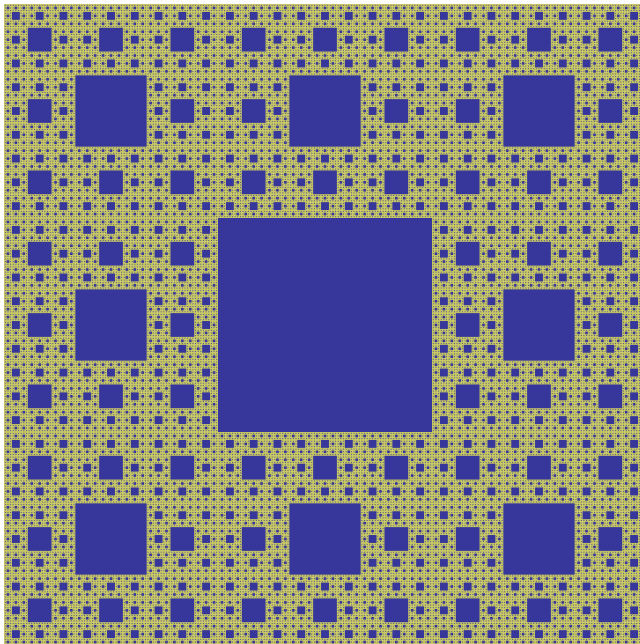
Some alternative punching methods are commented out.
"""
s = step // 3          # the new step
# iterate through the 9 sub-squares
# the central one will be filled with the color
for i in range(3):
    for j in range(3):
        if i != j or i != 1: # this is not the central cube
            if s >= 3:      # recursing needed?
                punch(x+i*s, y+j*s, s)      # recurse
            else:          # punching alternatives are:
                pm.setRect((x+s, y+s, x+2*s, y+2*s), color) # fill with a color
                #pm.copyPixmap(fill, (x+s, y+s, x+2*s, y+2*s)) # copy from fill
                #pm.invertIRect((x+s, y+s, x+2*s, y+2*s)) # invert colors

return

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

```

The result should look something like this:



4.1.13 How to Interface with NumPy

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

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

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

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

4.1.14 How to Add Images to a PDF Page

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

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

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

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

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

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

4.2 Text

4.2.1 How to Extract all Document Text

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

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

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

```
import sys, fitz
fname = sys.argv[1] # get document filename
doc = fitz.open(fname) # open document
out = open(fname + ".txt", "wb") # open text output
for page in doc: # iterate the document pages
    text = page.getText().encode("utf8") # get plain text (is in UTF-8)
    out.write(text) # write text of page
    out.write(bytes((12,))) # write page delimiter (form feed 0x0C)
out.close()
```

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

You have many options to cure this – see chapter [Appendix 2: Details on Text Extraction](#). Among them are:

1. Extract text in HTML format and store it as a HTML document, so it can be viewed in any browser.

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

See the following two section for examples and further explanations.

4.2.2 How to Extract Text from within a Rectangle

There is now (v1.18.0) more than one way to achieve this. We therefore have created a [folder](#) in the PyMuPDF-Utilities repository specifically dealing with this topic.

4.2.3 How to Extract Text in Natural Reading Order

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

    Notes:
        Method 'getTextWords()' does not try to recover words, if their single
        letters do not appear in correct lexical order. This function steps in
        here and creates a new list of recovered words.

    Args:
        words: list of words as created by 'getTextWords()'
        rect: rectangle to consider (usually the full page)

    Returns:
        List of recovered words. Same format as 'getTextWords', but left out
        block, line and word number - a list of items of the following format:
        [x0, y0, x1, y1, "word"]
    """
    # build my sublist of words contained in given rectangle
    mywords = [w for w in words if fitz.Rect(w[:4]) in rect]

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

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

    words_out = [] # we will return this

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

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

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

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

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

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

    return words_out

```

(continues on next page)

(continued from previous page)

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

    Notes:
        Can be adjusted / extended in obvious ways, e.g. using regular
        expressions, or being case insensitive, or only looking for complete
        words, etc.

    Args:
        text: string to be searched for
        words: list of items in format delivered by 'getTextWords()'.

    Returns:
        List of rectangles, one for each found locations.
    """
    rect_list = []
    for w in words:
        if text in w[4]:
            rect_list.append(fitz.Rect(w[:4]))

    return rect_list
```

4.2.4 How to Extract Tables from Documents

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

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

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

4.2.5 How to Search for and Mark Text

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

This method has advantages and drawbacks. Pros are

- The search string can contain blanks and wrap across lines
- Upper or lower case characters are treated equal
- Word hyphenation at line ends is detected and resolved
- return may also be a list of `Quad` objects to precisely locate text that is **not parallel** to either axis.

But you also have other options:

```
import sys
import fitz
```

(continues on next page)

(continued from previous page)

```

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

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

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

new_doc = False                           # indicator if anything found at all

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

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

```

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

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

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

PyMuPDF is hosted on GitHub³. We also are registered on PyPI⁴.

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

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

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

-
- ¹ <http://www.mupdf.com/>
 - ² <http://www.sumatrapdfreader.org/>
 - ³ <https://github.com/rk700/PyMuPDF>
 - ⁴ <https://pypi.org/project/PyMuPDF/>

4.2.6 How to Analyze Font Characteristics

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

```
import fitz

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

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

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

(continues on next page)

(continued from previous page)

```

        s["size"], # font size
        s["color"], # font color
    )
    print("Text: '%s'" % s["text"]) # simple print of text
    print(font_properties)

```

Here is the PDF page and the script output:

```

Text using fontname 'cour'
Text using fontname 'coit'
Text using fontname 'cobo'
Text using fontname 'cobi'
Text using fontname 'tiro'
Text using fontname 'titi'
Text using fontname 'tibo'
Text using fontname 'tibi'
Text using fontname 'helv'
Text using fontname 'heif'
Text using fontname 'hebo'
Text using fontname 'hebi'
*!▼◆▲※※※□■▼■●○※△■●※○△
Τεξτ υσινγ φοντναμε εσινμβι
Text using fontname 'china-s': 我很喜欢德国! 德国是个好地方!
Text using fontname 'china-t': 我很喜欢德国! 德国是个好地方!
Text using fontname 'japan': 世紀末以降における熊野三山
Text using fontname 'korea': 에듀툼은 하나의 계정으로

Text using fontname 'cour'
Font: 'Courier' (sans, monospaced), size 11, color #000000

Text using fontname 'coit'
Font: 'Courier-Oblique' (italic, sans, monospaced), size 11, color #ff0000

Text using fontname 'cobo'
Font: 'Courier-Bold' (sans, monospaced, bold), size 11, color #00ff00

Text using fontname 'cobl'
Font: 'Courier-BoldOblique' (italic, sans, monospaced, bold), size 11, color #0000ff

Text using fontname 'tiro'
Font: 'Times-Roman' (serifed, proportional), size 11, color #000000

Text using fontname 'titi'
Font: 'Times-Italic' (italic, serifed, proportional), size 11, color #ff0000

Text using fontname 'tibo'
Font: 'Times-Bold' (serifed, proportional, bold), size 11, color #00ff00

Text using fontname 'tibi'
Font: 'Times-BoldItalic' (italic, serifed, proportional, bold), size 11, color #0000ff

Text using fontname 'helv'
Font: 'Helvetica' (sans, proportional), size 11, color #000000

Text using fontname 'heif'
Font: 'Helvetica-Oblique' (italic, sans, proportional), size 11, color #ff0000

Text using fontname 'hebo'
Font: 'Helvetica-Bold' (sans, proportional, bold), size 11, color #00ff00

Text using fontname 'hebi'
Font: 'Helvetica-BoldOblique' (italic, sans, proportional, bold), size 11, color #0000ff

Text using fontname 'zadb'
Font: 'ZapfDingbats' (sans, proportional), size 11, color #000000

Text using fontname 'symb'
Font: 'Symbol' (sans, proportional), size 11, color #ff0000

Text using fontname 'china-s': 我很喜欢德国! 德国是个好地方!
Font: 'Heiti' (sans, proportional), size 11, color #00ff00

Text using fontname 'china-t': 我很喜欢德国! 德国是个好地方!
Font: 'Fangti' (sans, proportional), size 11, color #0000ff

Text using fontname 'japan': 世紀末以降における熊野三山
Font: 'Gothic' (sans, proportional), size 11, color #000000

Text using fontname 'korea': 에듀툼은 하나의 계정으로
Font: 'Dotum' (sans, proportional), size 11, color #ff0000

```

4.2.7 How to Insert Text

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

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

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

- `Page.insertFont()` – install a font for the page for later reference. The result is reflected in the output of `Document.getPageFontList()`. The font can be:
 - provided as a file,
 - already present somewhere in **this or another** PDF, or
 - be a **built-in** font.
- `Page.insertText()` – write some lines of text. Internally, this uses `Shape.insertText()`.
- `Page.insertTextbox()` – fit text in a given rectangle. Here you can choose text alignment features (left, right, centered, justified) and you keep control as to whether text actually fits. Internally, this uses `Shape.insertTextbox()`.

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

4.2.7.1 How to Write Text Lines

Output some text lines on a page:

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

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

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

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

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

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

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

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

```
import fitz
doc = fitz.open()
page = doc.newPage()
# the text strings, each having 3 lines
text1 = "rotate=0\nLine 2\nLine 3"
text2 = "rotate=90\nLine 2\nLine 3"
```

(continues on next page)

(continued from previous page)

```

text3 = "rotate=-90\nLine 2\nLine 3"
text4 = "rotate=180\nLine 2\nLine 3"
red = (1, 0, 0) # the color for the red dots
# the insertion points, each with a 25 pix distance from the corners
p1 = fitz.Point(25, 25)
p2 = fitz.Point(page.rect.width - 25, 25)
p3 = fitz.Point(25, page.rect.height - 25)
p4 = fitz.Point(page.rect.width - 25, page.rect.height - 25)
# create a Shape to draw on
shape = page.newShape()

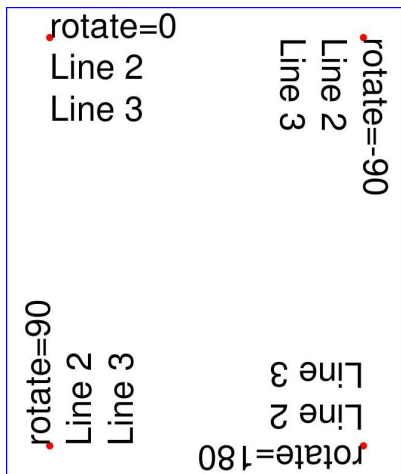
# draw the insertion points as red, filled dots
shape.drawCircle(p1,1)
shape.drawCircle(p2,1)
shape.drawCircle(p3,1)
shape.drawCircle(p4,1)
shape.finish(width=0.3, color=red, fill=red)

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

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

```

This is the result:



4.2.7.2 How to Fill a Text Box

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

```

import fitz
doc = fitz.open(...) # new or existing PDF
page = doc.newPage() # new page, or choose doc[n]

```

(continues on next page)

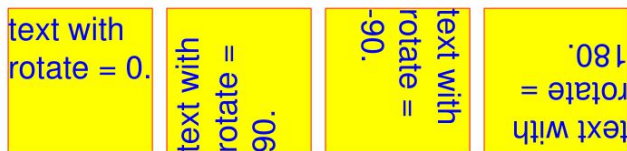
(continued from previous page)

```

r1 = fitz.Rect(50,100,100,150) # a 50x50 rectangle
disp = fitz.Rect(55, 0, 55, 0) # add this to get more rects
r2 = r1 + disp # 2nd rect
r3 = r1 + disp * 2 # 3rd rect
r4 = r1 + disp * 3 # 4th rect
t1 = "text with rotate = 0." # the texts we will put in
t2 = "text with rotate = 90."
t3 = "text with rotate = -90."
t4 = "text with rotate = 180."
red = (1,0,0) # some colors
gold = (1,1,0)
blue = (0,0,1)
"""We use a Shape object (something like a canvas) to output the text and
the rectangles surrounding it for demonstration.
"""
shape = page.newShape() # create Shape
shape.drawRect(r1) # draw rectangles
shape.drawRect(r2) # giving them
shape.drawRect(r3) # a yellow background
shape.drawRect(r4) # and a red border
shape.finish(width = 0.3, color = red, fill = gold)
# Now insert text in the rectangles. Font "Helvetica" will be used
# by default. A return code rc < 0 indicates insufficient space (not checked here).
rc = shape.insertTextbox(r1, t1, color = blue)
rc = shape.insertTextbox(r2, t2, color = blue, rotate = 90)
rc = shape.insertTextbox(r3, t3, color = blue, rotate = -90)
rc = shape.insertTextbox(r4, t4, color = blue, rotate = 180)
shape.commit() # write all stuff to page /Contents
doc.save("...")

```

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



4.2.7.3 How to Use Non-Standard Encoding

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

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

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

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

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

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

The result:

Sômé tèxt wìth nŏñ-Lâtîn characterß.

Стми тѳxt wμth нѳр-Lβтξη characterι.

STmИ tXxt wЛth нЖЯ-ЛБтHη characterЪ.

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

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

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

4.3 Annotations

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

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

- MuPDF now provides its own annotation icons where relevant. PyMuPDF switched to using them (for ‘FileAttachment’ and ‘Text’ [“sticky note”] so far).
- MuPDF now also supports ‘Caret’, ‘Movie’, ‘Sound’ and ‘Signature’ annotations, which we may include in PyMuPDF at some later time.

4.3.1 How to Add and Modify Annotations

In PyMuPDF, new annotations can be added via *Page* methods. Once an annotation exists, it can be modified to a large extent using methods of the *Annot* class.

In contrast to many other tools, initial insert of annotations happens with a minimum number of properties. We leave it to the programmer to e.g. set attributes like author, creation date or subject.

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

```
# -*- coding: utf-8 -*-
"""
-----
Demo script showing how annotations can be added to a PDF using PyMuPDF.

It contains the following annotation types:
Caret, Text, FreeText, text markers (underline, strike-out, highlight,
squiggle), Circle, Square, Line, PolyLine, Polygon, FileAttachment, Stamp
and Redaction.
There is some effort to vary appearances by adding colors, line ends,
opacity, rotation, dashed lines, etc.

Dependencies
-----
PyMuPDF v1.17.0
-----
"""
from __future__ import print_function

import gc
import os
import sys

import fitz

print(fitz.__doc__)
if fitz.VersionBind.split(".") < ["1", "17", "0"]:
    sys.exit("PyMuPDF v1.17.0+ is needed.")

gc.set_debug(gc.DEBUG_UNCOLLECTABLE)

highlight = "this text is highlighted"
underline = "this text is underlined"
strikeout = "this text is striked out"
squiggled = "this text is zigzag-underlined"
red = (1, 0, 0)
blue = (0, 0, 1)
gold = (1, 1, 0)
green = (0, 1, 0)
```

(continues on next page)

(continued from previous page)

```

displ = fitz.Rect(0, 50, 0, 50)
r = fitz.Rect(72, 72, 220, 100)
t1 = u"têxt üsès Lätĩñ charß,\nEUR: €, mu: µ, super scripts: 23!"

def print_descr(annot):
    """Print a short description to the right of each annot rect."""
    annot.parent.insertText(
        annot.rect.br + (10, -5), "%s annotation" % annot.type[1], color=red
    )

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

page.setRotation(0)

annot = page.addCaretAnnot(r.tl)
print_descr(annot)

r = r + displ
annot = page.addFreetextAnnot(
    r,
    t1,
    fontsize=10,
    rotate=90,
    text_color=blue,
    fill_color=gold,
    align=fitz.TEXT_ALIGN_CENTER,
)
annot.setBorder(width=0.3, dashes=[2])
annot.update(text_color=blue, fill_color=gold)

print_descr(annot)
r = annot.rect + displ

annot = page.addTextAnnot(r.tl, t1)
print_descr(annot)

# Adding text marker annotations:
# first insert a unique text, then search for it, then mark it
pos = annot.rect.tl + displ.tl
page.insertText(
    pos, # insertion point
    highlight, # inserted text
    morph=(pos, fitz.Matrix(-5)), # rotate around insertion point
)
rl = page.searchFor(highlight, quads=True) # need a quad b/o tilted text
annot = page.addHighlightAnnot(rl[0])
print_descr(annot)
pos = annot.rect.bl # next insertion point

page.insertText(pos, underline, morph=(pos, fitz.Matrix(-10)))
rl = page.searchFor(underline, quads=True)
annot = page.addUnderlineAnnot(rl[0])
print_descr(annot)
pos = annot.rect.bl

```

(continues on next page)

(continued from previous page)

```

page.insertText(pos, strikeouts, morph=(pos, fitz.Matrix(-15)))
rl = page.searchFor(strikeout, quads=True)
annot = page.addStrikeoutAnnot(rl[0])
print_descr(annot)
pos = annot.rect.bl

page.insertText(pos, squiggled, morph=(pos, fitz.Matrix(-20)))
rl = page.searchFor(squiggled, quads=True)
annot = page.addSquigglyAnnot(rl[0])
print_descr(annot)
pos = annot.rect.bl

r = fitz.Rect(pos, pos.x + 75, pos.y + 35) + (0, 20, 0, 20)
annot = page.addPolylineAnnot([r.bl, r.tr, r.br, r.tl]) # 'Polyline'
annot.setBorder(width=0.3, dashes=[2])
annot.setColors(stroke=blue, fill=green)
annot.setLineEnds(fitz.PDF_ANNOT_LE_CLOSED_ARROW, fitz.PDF_ANNOT_LE_R_CLOSED_ARROW)
annot.update(fill_color=(1, 1, 0))
print_descr(annot)

r += displ
annot = page.addPolygonAnnot([r.bl, r.tr, r.br, r.tl]) # 'Polygon'
annot.setBorder(width=0.3, dashes=[2])
annot.setColors(stroke=blue, fill=gold)
annot.setLineEnds(fitz.PDF_ANNOT_LE_DIAMOND, fitz.PDF_ANNOT_LE_CIRCLE)
annot.update()
print_descr(annot)

r += displ
annot = page.addLineAnnot(r.tr, r.bl) # 'Line'
annot.setBorder(width=0.3, dashes=[2])
annot.setColors(stroke=blue, fill=gold)
annot.setLineEnds(fitz.PDF_ANNOT_LE_DIAMOND, fitz.PDF_ANNOT_LE_CIRCLE)
annot.update()
print_descr(annot)

r += displ
annot = page.addRectAnnot(r) # 'Square'
annot.setBorder(width=1, dashes=[1, 2])
annot.setColors(stroke=blue, fill=gold)
annot.update(opacity=0.5)
print_descr(annot)

r += displ
annot = page.addCircleAnnot(r) # 'Circle'
annot.setBorder(width=0.3, dashes=[2])
annot.setColors(stroke=blue, fill=gold)
annot.update()
print_descr(annot)

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

```

(continues on next page)

(continued from previous page)

```
r += displ
annot = page.addStampAnnot(r, stamp=10) # 'Stamp'
annot.setColors(stroke=green)
annot.update()
print_descr(annot)

r += displ + (0, 0, 50, 10)
rc = page.insertTextbox(
    r,
    "This content will be removed upon applying the redaction.",
    color=blue,
    align=fitz.TEXT_ALIGN_CENTER,
)
annot = page.addRedactAnnot(r)
print_descr(annot)

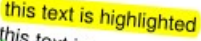
outfile = os.path.abspath(__file__).replace(".py", "-%i.pdf" % page.rotation)
doc.save(outfile, deflate=True)
```

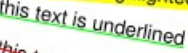
This script should lead to the following output:

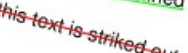
 Caret annotation

 FreeText annotation

 Text annotation

 this text is highlighted Highlight annotation

 this text is underlined Underline annotation


 this text is ~~striked-out~~ StrikeOut annotation

 this text is zigzag-underlined Squiggly annotation

 PolyLine annotation

 Polygon annotation

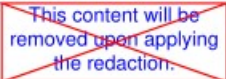
 Line annotation

 Square annotation

 Circle annotation

 FileAttachment annotation

 NOT FOR PUBLIC RELEASE Stamp annotation

 This content will be removed upon applying the redaction. Redact annotation

4.3.2 How to Mark Text

This script searches for text and marks it:

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

The result looks like this:



4.3.3 How to Use FreeText

This script shows a couple of ways to deal with ‘FreeText’ annotations:

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

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

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

(continues on next page)

(continued from previous page)

```

page = doc.newPage()

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

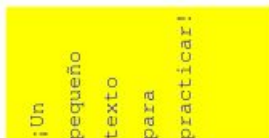
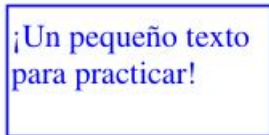
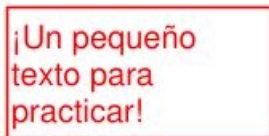
# the text, Latin alphabet
t = "¡Un pequeño texto para practicar!"

# add 3 annots, modify the last one somewhat
a1 = page.addFreetextAnnot(r1, t, color=red)
a2 = page.addFreetextAnnot(r2, t, fontname="Ti", color=blue)
a3 = page.addFreetextAnnot(r3, t, fontname="Co", color=blue, rotate=90)
a3.setBorder(width=0)
a3.update(fontsize=8, fill_color=gold)

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

```

The result looks like this:



4.3.4 Using Buttons and JavaScript

Since MuPDF v1.16, ‘FreeText’ annotations no longer support bold or italic versions of the Times-Roman, Helvetica or Courier fonts.

A big **thank you** to our user [@kurokawaikki](#), who contributed the following script to **circumvent this restriction**.

```

"""
Problem: Since MuPDF v1.16 a 'Freetext' annotation font is restricted to the
"normal" versions (no bold, no italics) of Times-Roman, Helvetica, Courier.
It is impossible to use PyMuPDF to modify this.

```

(continues on next page)

(continued from previous page)

*Solution: Using Adobe's JavaScript API, it is possible to manipulate properties of Freetext annotations. Check out these references:
https://www.adobe.com/content/dam/acom/en/devnet/acrobat/pdfs/js_api_reference.pdf,
 or <https://www.adobe.com/devnet/acrobat/documentation.html>.*

Function 'this.getAnnots()' will return all annotations as an array. We loop over this array to set the properties of the text through the 'richContents' attribute.

There is no explicit property to set text to bold, but it is possible to set `fontWeight=800` (400 is the normal size) of richContents.

Other attributes, like color, italics, etc. can also be set via richContents.

If we have 'FreeText' annotations created with PyMuPDF, we can make use of this JavaScript feature to modify the font - thus circumventing the above restriction.

Use PyMuPDF v1.16.12 to create a push button that executes a Javascript containing the desired code. This is what this program does.

Then open the resulting file with Adobe reader (!).

After clicking on the button, all Freetext annotations will be bold, and the file can be saved.

If desired, the button can be removed again, using free tools like PyMuPDF or PDF XChange editor.

Note / Caution:

*The JavaScript will **only** work if the file is opened with Adobe Acrobat reader! When using other PDF viewers, the reaction is unforeseeable.*

"""

```
import sys
```

```
import fitz
```

```
# this JavaScript will execute when the button is clicked:
```

```
jscript = """
var annt = this.getAnnots();
annt.forEach(function (item, index) {
    try {
        var span = item.richContents;
        span.forEach(function (it, dx) {
            it.fontWeight = 800;
        })
        item.richContents = span;
    } catch (err) {}
});
app.alert('Done');
"""
```

```
i_fn = sys.argv[1] # input file name
o_fn = "bold-" + i_fn # output filename
doc = fitz.open(i_fn) # open input
page = doc[0] # get desired page
```

```
# -----
# make a push button for invoking the JavaScript
# -----
```

```
widget = fitz.Widget() # create widget
```

(continues on next page)

(continued from previous page)

```

# make it a 'PushButton'
widget.field_type = fitz.PDF_WIDGET_TYPE_BUTTON
widget.field_flags = fitz.PDF_BTN_FIELD_IS_PUSHBUTTON

widget.rect = fitz.Rect(5, 5, 20, 20) # button position

widget.script = jscript # fill in JavaScript source text
widget.field_name = "Make bold" # arbitrary name
widget.field_value = "Off" # arbitrary value
widget.fill_color = (0, 0, 1) # make button visible

annot = page.addWidget(widget) # add the widget to the page
doc.save(o_fn) # output the file

```

4.3.5 How to Use Ink Annotations

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

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

```

import math
import fitz

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

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

```

(continues on next page)

(continued from previous page)

```

page = doc.newPage() # give it a page

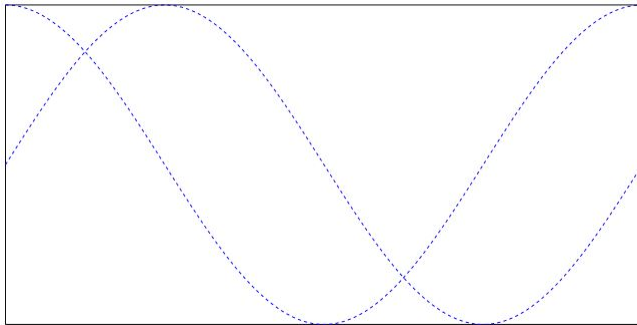
#-----
# add the Ink annotation, consisting of 2 curve segments
#-----
annot = page.addInkAnnot((sin_points, cos_points))
# let it look a little nicer
annot.setBorder(width=0.3, dashes=[1,]) # line thickness, some dashing
annot.setColors(stroke=(0,0,1)) # make the lines blue
annot.update() # update the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

page.drawRect(rect, width=0.3) # only to demonstrate we did OK

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

```

This is the result:



4.4 Drawing and Graphics

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

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

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

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

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

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

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

```

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

@author: Jorj
@license: GNU GPL 3.0+

Create a list of available symbols defined in shapes_and_symbols.py

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

"""

import fitz
import shapes_and_symbols as sas

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

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

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

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

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

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

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

```

This is the script's outcome:

-  arrow (easy)
-  caro (easy)
-  clover (easy)
-  diamond (easy)
-  do not enter (medium)
-  frowney (medium)
-  hand (complex)
-  heart (easy)
-  pencil (very complex)
-  smiley (easy)

4.5 Extracting Drawings

(New in v1.18.0)

The drawing commands issued by a page can be extracted. Interestingly, this is possible for **all supported document types** – not just PDF: so you can use it for XPS, EPUB and others as well.

A new page method, `Page.getDrawings()` accesses draw commands and converts them into a list of Python dictionaries. Each dictionary – called a “path” – represents a separate drawing – it may be simple like a single line, or a complex combination of lines and curves representing one of the shapes of the previous section.

The *path* dictionary has been designed such that it can easily be used by the *Shape* class and its methods.

The following is a code snippet which extracts the drawings of a page and re-draws them on a new page:

```
import fitz
doc = fitz.open("some.file")
page = doc[0]
paths = page.getDrawings() # extract existing drawings
# this is a list of "paths", which can directly be drawn again using Shape
# -----
#
# define some output page with the same dimensions
outpdf = fitz.open()
outpage = outpdf.newPage(width=page.rect.width, height=page.rect.height)
shape = outpage.newShape() # make a drawing canvas for the output page
# -----
# loop through the paths and draw them
# -----
```

(continues on next page)

(continued from previous page)

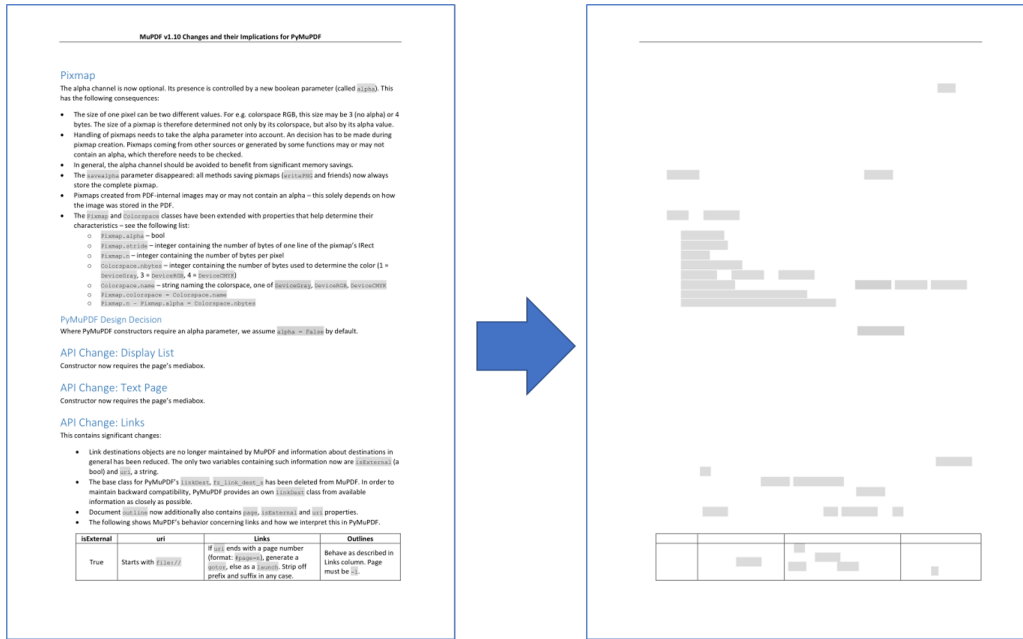
```

for path in paths:
    # -----
    # draw each entry of the 'items' list
    # -----
    for item in path["items"]: # these are the draw commands
        if item[0] == "l": # line
            shape.drawLine(item[1], item[2])
        elif item[0] == "re": # rectangle
            shape.drawRect(item[1])
        elif item[0] == "c": # curve
            shape.drawBezier(item[1], item[2], item[3], item[4])
        else:
            raise ValueError("unhandled drawing", item)
    # -----
    # all items are drawn, now apply the common properties
    # to finish the path
    # -----
    shape.finish(
        fill=path["fill"], # fill color
        color=path["color"], # line color
        dashes=path["dashes"], # line dashing
        even_odd=path["even_odd"], # control color of overlaps
        closePath=path["closePath"], # whether to connect last and first point
        lineJoin=path["lineJoin"], # how line joins should look like
        lineCap=max(path["lineCap"]), # how line ends should look like
        width=path["width"], # line width
        stroke_opacity=path["opacity"], # same value for both
        fill_opacity=path["opacity"], # opacity parameters
    )
# all paths processed - commit the shape to its page
shape.commit()
outpdf.save("drawings-page-0.pdf")

```

As can be seen, there is a high congruence level with the *Shape* class. With one exception: For technical reasons `lineCap` is a tuple of 3 numbers here, whereas it is an integer in *Shape* (and in PDF). So we simply take the maximum value of that tuple.

Here is a comparison between input and output of an example page, created by the previous script:



Note: The reconstruction of graphics like shown here is not perfect. The following aspects will not be reproduced as of this version:

- Page definitions can be complex and include instructions for not showing / hiding certain areas to keep them invisible. Things like this are ignored by `Page.getDrawings()` - it will always return all paths.

Note: You can use the path list to make your own lists of e.g. all lines or all rectangles on the page, subselect them by criteria like color or position on the page etc.

4.6 Multiprocessing

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

No Python threading support.

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

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

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

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

Depending on the number of CPUs, the document is divided in page ranges.
Each range is then worked on by one process.
The type of work would typically be text extraction or page rendering. Each
```

(continues on next page)

(continued from previous page)

```

process must know where to put its results, because this processing pattern
does not include inter-process communication or data sharing.

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

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

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

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

    Args:
        vector: a list containing required parameters.
    """
    # recreate the arguments
    idx = vector[0] # this is the segment number we have to process
    cpu = vector[1] # number of CPUs
    filename = vector[2] # document filename
    mat = vector[3] # the matrix for rendering
    doc = fitz.open(filename) # open the document
    num_pages = len(doc) # get number of pages

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

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

```

(continues on next page)

(continued from previous page)

```

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

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

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

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

```

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

```

"""
Created on 2019-05-01

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

Demonstrate the use of multiprocessing with PyMuPDF
-----
This example shows some more advanced use of multiprocessing.
The main process show a Qt GUI and establishes a 2-way communication with
another process, which accesses a supported document.
"""
import os
import sys
import time
import multiprocessing as mp
import queue
import fitz
from PyQt5 import QtCore, QtGui, QtWidgets

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

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

```

(continues on next page)

(continued from previous page)

```

self.timerGet.timeout.connect(self.onTimerGetPage)
self.timerWaiting = QtCore.QTimer(self)
self.timerWaiting.timeout.connect(self.onTimerWaiting)
self.initUI()

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

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

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

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

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

    vbox.addLayout(hbox)

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

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

def openDoc(self):
    path, _ = QtWidgets.QFileDialog.getOpenFileName(
        self,
        "Open Document",
        self.lastDir,
        "All Supported Files (*.pdf;*.epub;*.xps;*.oxps;*.cbz;*.fb2);;PDF Files_↵
↵(*.pdf);;EPUB Files (*.epub);;XPS Files (*.xps);;OpenXPS Files (*.oxps);;CBZ Files_↵
↵(*.cbz);;FB2 Files (*.fb2)",
        options=QtWidgets.QFileDialog.Options(),
    )
    if path:
        self.lastDir, self.file = os.path.split(path)
        if self.process:
            self.queueNum.put(-1) # use -1 to notify the process to exit
        self.timerSend.stop()
        self.curPageNum = 0
        self.pageCount = 0
        self.process = mp.Process(

```

(continues on next page)

(continued from previous page)

```

        target=openDocInProcess, args=(path, self.queNum, self.queDoc)
    )
    self.process.start()
    self.timerGet.start(40)
    self.label.setText("0/0")
    self.queNum.put(0)
    self.startTime = time.perf_counter()
    self.timerWaiting.start(40)

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

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

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

def onTimerGetPage(self):
    try:
        ret = self.queDoc.get(False)
        if isinstance(ret, int):
            self.timerWaiting.stop()
            self.pageCount = ret
            self.label.setText("{} / {}".format(self.curPageNum + 1, self.
↪pageCount))
        else: # tuple, pixmap info
            num, samples, width, height, stride, alpha = ret
            self.curPageNum = num
            self.label.setText("{} / {}".format(self.curPageNum + 1, self.
↪pageCount))

            fmt = (
                QtGui.QImage.Format_RGBA8888
                if alpha
                else QtGui.QImage.Format_RGB888
            )
            qimg = QtGui.QImage(samples, width, height, stride, fmt)
            self.labelImg.setPixmap(QtGui.QPixmap.fromImage(qimg))
    except queue.Empty as ex:
        pass

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

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

def openDocInProcess(path, queNum, quePageInfo):

```

(continues on next page)

(continued from previous page)

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

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

4.7 General

4.7.1 How to Open with a Wrong File Extension

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

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

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

Note: MuPDF itself does not try to determine the file type from the file contents. **You** are responsible for supplying the filetype info in some way – either implicitly via the file extension, or explicitly as shown. There are pure Python packages like `filetype` that help you doing this. Also consult the *Document* chapter for a full description.

4.7.2 How to Embed or Attach Files

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

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

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

There exist several example scripts: [embedded-list.py](#), [new-annots.py](#).

Also look at the sections above and at chapter [Appendix 3: Considerations on Embedded Files](#).

4.7.3 How to Delete and Re-Arrange Pages

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

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

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

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

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

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

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

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

For more information also have a look at this [Wiki article](#).

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

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

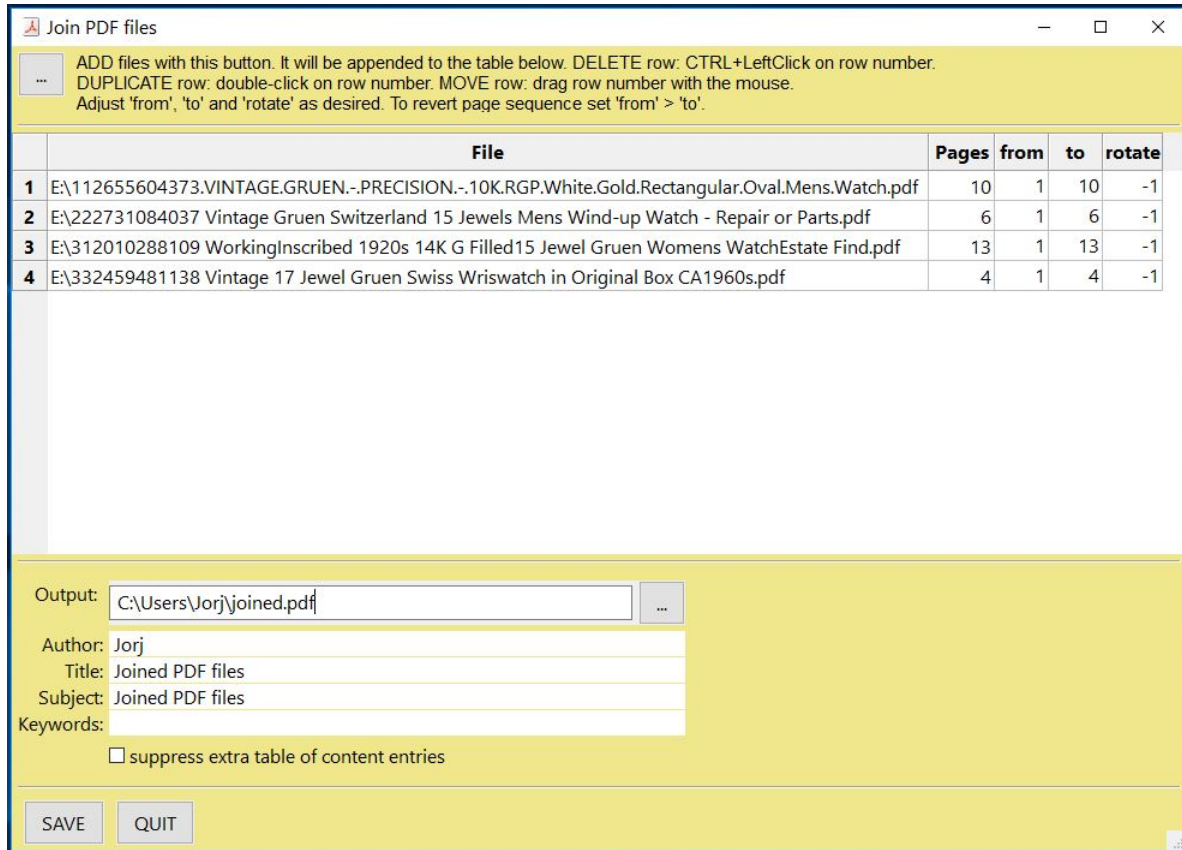
This snippet duplicates the PDF with itself so that it will contain the pages $0, 1, \dots, n, 0, 1, \dots, n$ (**extremely fast and without noticeably increasing the file size!**):

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

4.7.4 How to Join PDFs

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

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



4.7.5 How to Add Pages

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

newPage

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

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

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

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

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

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

The convenience function `PaperSize()` knows over 40 industry standard paper formats to choose from. To see them, inspect dictionary `paperSizes`. Pass the desired dictionary key to `PaperSize()` to retrieve the paper dimensions. Upper and lower case is supported. If you append “-L” to the format name, the landscape version is returned.

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

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

insertPage

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

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

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

4.7.6 How To Dynamically Clean Up Corrupt PDFs

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

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

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

#-----
# 'Tolerant' PDF reader
#-----
def reader(fname, password = None):
    idata = open(fname, "rb").read() # read the PDF into memory and
```

(continues on next page)

(continued from previous page)

```

ibuffer = BytesIO(idata) # convert to stream
if password is None:
    try:
        return PdfReader(ibuffer) # if this works: fine!
    except:
        pass

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

```

With the command line utility *pdftk* (available for Windows only, but reported to also run under *Wine*) 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.

4.7.7 How to Split Single Pages

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

```

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

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

for spage in src: # for each page in input
    r = spage.rect # input page rectangle
    d = fitz.Rect(spage.CropBoxPosition, # CropBox displacement if not
                 spage.CropBoxPosition) # starting at (0, 0)

    #-----
    # example: cut input page into 2 x 2 parts
    #-----

    r1 = r * 0.5 # top left rect
    r2 = r1 + (r1.width, 0, r1.width, 0) # top right rect
    r3 = r1 + (0, r1.height, 0, r1.height) # bottom left rect
    r4 = fitz.Rect(r1.br, r.br) # bottom right rect
    rect_list = [r1, r2, r3, r4] # put them in a list

    for rx in rect_list: # run thru rect list
        rx += d # add the CropBox displacement
        page = doc.newPage(-1, # new output page with rx dimensions
                          width = rx.width,
                          height = rx.height)

        page.showPDFpage (
            page.rect, # fill all new page with the image
            src, # input document
            spage.number, # input page number
            clip = rx, # which part to use of input page
        )

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

This shows what happens to an input page:



4.7.8 How to Combine Single Pages

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
src, # input document
page.number) # input page number

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

Example effect:



4.7.9 How to Convert Any Document to PDF

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

doc = fitz.open(fn)

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

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

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

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

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

```

4.7.10 How to Deal with Messages Issued by MuPDF

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

In addition, these messages go to the internal buffer together with any MuPDF warnings – see below.

We always prefix these messages with an identifying string `"mupdf:"`. If you prefer to not see recoverable MuPDF errors at all, issue the command `fitz.TOOLS.mupdf_display_errors(False)`.

MuPDF warnings continue to be stored in an internal buffer and can be viewed using `Tools.mupdf_warnings()`.

Please note that MuPDF errors may or may not lead to Python exceptions. In other words, you may see error messages from which MuPDF can recover and continue processing.

Example output for a **recoverable error**. We are opening a damaged PDF, but MuPDF is able to repair it and gives us a few information on what happened. Then we illustrate how to find out whether the document can later be saved incrementally. Checking the `Document.isDirty` attribute at this point also indicates that the open had to repair the document:

```
>>> import fitz
>>> doc = fitz.open("damaged-file.pdf") # leads to a sys.stderr message:
mupdf: cannot find startxref
>>> print(fitz.TOOLS.mupdf_warnings()) # check if there is more info:
cannot find startxref
trying to repair broken xref
repairing PDF document
object missing 'endobj' token
>>> doc.can_save_incrementally() # this is to be expected:
False
>>> # the following indicates whether there are updates so far
>>> # this is the case because of the repair actions:
>>> doc.isDirty
True
>>> # the document has nevertheless been created:
>>> doc
fitz.Document('damaged-file.pdf')
>>> # we now know that any save must occur to a new file
```

Example output for an **unrecoverable error**:

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

4.7.11 How to Deal with PDF Encryption

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

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

Note: A PDF document may have two different passwords:

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

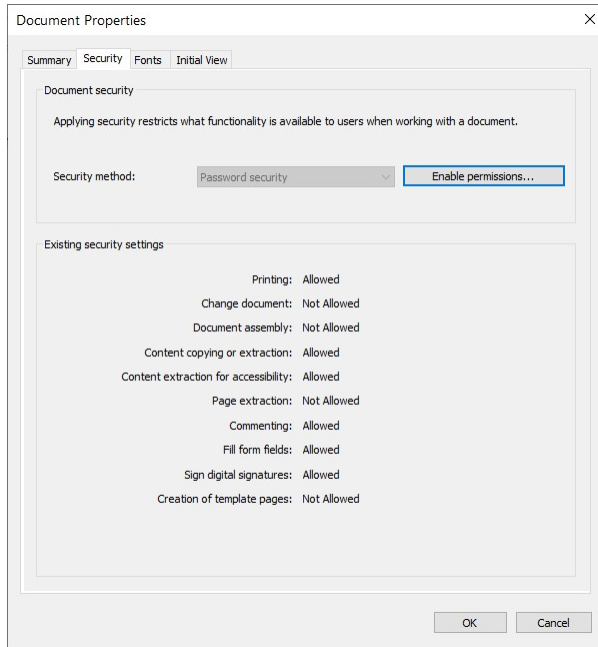
Method `Document.authenticate()` will automatically establish access rights according to the password used.

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

```
import fitz

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

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



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

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

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

4.8 Common Issues and their Solutions

4.8.1 Changing Annotations: Unexpected Behaviour

4.8.1.1 Problem

There are two scenarios:

1. **Updating** an annotation with PyMuPDF which was created by some other software.
2. **Creating** an annotation with PyMuPDF and later changing it with some other software.

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

4.8.1.2 Cause

Annotation maintenance is handled differently by each PDF maintenance application. Some annotation types may not be supported, or not be supported fully or some details may be handled in a different way than in another application. **There is no standard.**

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

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

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

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

4.8.1.3 Solutions

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

1. Stay with the same software for **creating and changing** an annotation.
2. When using PyMuPDF to change an “alien” annotation, try to **avoid** `Annot.update()`. The following methods **can be used without it**, so that the original appearance should be maintained:
 - `Annot.set_rect()` (location changes)
 - `Annot.set_flags()` (annotation behaviour)
 - `Annot.set_info()` (meta information, except changes to *content*)
 - `Annot.set_popup()` (create popup or change its rect)
 - `Annot.set_optional_content()` (add / remove reference to optional content information)
 - `Annot.set_open()`
 - `Annot.update_file()` (file attachment changes)

4.8.2 Misplaced Item Insertions on PDF Pages

4.8.2.1 Problem

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

4.8.2.2 Cause

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

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

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

So the PDF creator did this:

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

where they should have done this:

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

Note:

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

4.8.2.3 Solutions

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

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

1. The easiest way: in your script, do a `Page.cleanContents()` before you do your first item insertion.
2. Pre-process your PDF with the MuPDF command line utility `mutool clean -c ...` and work with its output file instead.
3. Directly wrap the page’s *contents* with the stacking commands before you do your first item insertion.

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

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

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

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

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

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

4.9 Low-Level Interfaces

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

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

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

4.9.1 How to Iterate through the *xref* Table

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

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

This produces the following output:

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

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

object 3 (stream: False)
<<
```

(continues on next page)

```

    /Kids [ 4 0 R 5 0 R ]
    /Type /Pages
    /Count 2
>>

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

```

A PDF object definition is an ordinary ASCII string.

4.9.2 How to Handle Object Streams

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

Objects of these types are called “stream objects”. PyMuPDF allows reading an object’s stream via method `Document.xrefStream()` with the object’s `xref` as an argument. And it is also possible to write back a modified version of a stream using `Document.updateStream()`.

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

```

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

```

`Document.xrefStream()` automatically returns a stream decompressed as a bytes object – and `Document.updateStream()` automatically compresses it (where beneficial).

4.9.3 How to Handle Page Contents

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

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

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

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

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

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

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

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

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

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

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

4.9.4 How to Access the PDF Catalog

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

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

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

4.9.5 How to Access the PDF File Trailer

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

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

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

```
>>> import fitz
>>> doc=fitz.open("PyMuPDF.pdf")
>>> trailer=doc._getTrailerString()
>>> print(trailer)
<</Size 5535/Info 5275 0 R/Root 5274 0 R/ID[(\340\273fE\225^
↪1\226\2320|\003\201\325g\245) (#1, \317\205\000\371\251wO6\352Oa\021)]>>
>>>
```

4.9.6 How to Access XML Metadata

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

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

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

```

(continues on next page)

(continued from previous page)

```
omitted data
...
<?xpacket end="w"?>
```

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

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

Using *fitz* as a Module

(New in version 1.16.8)

PyMuPDF can also be used in the command line as a **module** to perform basic utility functions.

This is work in progress and subject to changes. This feature should obsolete writing some of the most basic scripts.

As a guideline we are using the feature set of MuPDF command line tools. Admittedly, there is some functional overlap. On the other hand, PDF embedded files are no longer supported by MuPDF, so PyMuPDF is offering something unique here.

5.1 Invocation

Invoke the module like this:

```
python -m fitz command parameters
```

General remarks:

- Request help via “-h”, resp. command-specific help via “*command -h*”.
- Parameters may be abbreviated as long as the result is not ambiguous (Python 3.5 or later only).
- Several commands support parameters *-pages* and *-xrefs*. They are intended for down-selection. Please note that:
 - **page numbers** for this utility must be given **1-based**.
 - valid *xref* numbers start at 1.
 - Specify any number of either single integers or integer ranges, separated by one comma each. A **range** is a pair of integers separated by one hyphen “-“. Integers must not exceed the maximum page number or resp. *xref* number. To specify that maximum, the symbolic variable “N” may be used instead of an integer. Integers or ranges may occur several times, in any sequence and may overlap. If in a range the first number is greater than the second one, the respective items will be processed in reversed order.
- You can also use the *fitz* module inside your script:

```

>>> from fitz.__main__ import main as fitz_command
>>> cmd = "clean input.pdf output.pdf -pages 1,N".split() # prepare command
>>> saved_parms = sys.argv[1:] # save original parameters
>>> sys.argv[1:] = cmd # store command
>>> fitz_command() # execute command
>>> sys.argv[1:] = saved_parms # restore original parameters

```

- You can use the following 2-liner and compile it with `Nuitka` in either normal or standalone mode, if you want to distribute it. This will give you a command line utility with all the functions explained below:

```

from fitz.__main__ import main
main()

```

5.2 Cleaning and Copying

This command will optimize the PDF and store the result in a new file. You can use it also for encryption, decryption and creating sub documents. It is mostly similar to the MuPDF command line utility “`mutool clean`”:

```

python -m fitz clean -h
usage: fitz clean [-h] [-password PASSWORD]
                [-encryption {keep,none,rc4-40,rc4-128,aes-128,aes-256}]
                [-owner OWNER] [-user USER] [-garbage {0,1,2,3,4}]
                [-compress] [-ascii] [-linear] [-permission PERMISSION]
                [-sanitize] [-pretty] [-pages PAGES]
                input output

----- optimize PDF or create sub-PDF if pages given -----

positional arguments:
input                  PDF filename
output                output PDF filename

optional arguments:
-h, --help            show this help message and exit
-password PASSWORD    password
-encryption {keep,none,rc4-40,rc4-128,aes-128,aes-256}
                    encryption method
-owner OWNER          owner password
-user USER           user password
-garbage {0,1,2,3,4}  garbage collection level
-compress             compress (deflate) output
-ascii               ASCII encode binary data
-linear              format for fast web display
-permission PERMISSION
                    integer with permission levels
-sanitize             sanitize / clean contents
-pretty              prettify PDF structure
-pages PAGES         output selected pages, format: 1,5-7,50-N

```

If you specify “-pages”, be aware that only page-related objects are copied, **no document-level items** like e.g. embedded files.

Please consult `Document.save()` for the parameter meanings.

5.3 Extracting Fonts and Images

Extract fonts or images from selected PDF pages to a desired directory:

```
python -m fitz extract -h
usage: fitz extract [-h] [-images] [-fonts] [-output OUTPUT] [-password PASSWORD]
                  [-pages PAGES]
                  input

----- extract images and fonts to disk -----

positional arguments:
input                  PDF filename

optional arguments:
-h, --help            show this help message and exit
-images               extract images
-fonts                extract fonts
-output OUTPUT        output directory, defaults to current
-password PASSWORD    password
-pages PAGES          only consider these pages, format: 1,5-7,50-N
```

Image filenames are built according to the naming scheme: “**img-xref.ext**”, where “ext” is the extension associated with the image and “xref” the *xref* of the image PDF object.

Font filenames consist of the fontname and the associated extension. Any spaces in the fontname are replaced with hyphens “-”.

The output directory must already exist.

Note: Except for output directory creation, this feature is **functionally equivalent** to and obsoletes [this script](#).

5.4 Joining PDF Documents

To join several PDF files specify:

```
python -m fitz join -h
usage: fitz join [-h] -output OUTPUT [input [input ...]]

----- join PDF documents -----

positional arguments:
input                  input filenames

optional arguments:
-h, --help            show this help message and exit
-output OUTPUT        output filename

specify each input as 'filename[,password[,pages]]'
```

Note:

1. Each input must be entered as “**filename,password,pages**”. Password and pages are optional.

2. The password entry **is required** if the “pages” entry is used. If the PDF needs no password, specify two commas.
3. The “**pages**” format is the same as explained at the top of this section.
4. Each input file is immediately closed after use. Therefore you can use one of them as output filename, and thus overwrite it.

Example: To join the following files

1. **file1.pdf**: all pages, back to front, no password
2. **file2.pdf**: last page, first page, password: “secret”
3. **file3.pdf**: pages 5 to last, no password

and store the result as **output.pdf** enter this command:

```
python -m fitz join -o output.pdf file1.pdf,,N-1 file2.pdf,secret,N,1 file3.pdf,,5-N
```

5.5 Low Level Information

Display PDF internal information. Again, there are similarities to “*mutool show*”:

```
python -m fitz show -h
usage: fitz show [-h] [-password PASSWORD] [-catalog] [-trailer] [-metadata]
               [-xrefs XREFS] [-pages PAGES]
               input

----- display PDF information -----

positional arguments:
input                  PDF filename

optional arguments:
-h, --help            show this help message and exit
-password PASSWORD    password
-catalog              show PDF catalog
-trailer              show PDF trailer
-metadata             show PDF metadata
-xrefs XREFS         show selected objects, format: 1,5-7,N
-pages PAGES         show selected pages, format: 1,5-7,50-N
```

Examples:

```
python -m fitz show x.pdf
PDF is password protected

python -m fitz show x.pdf -pass hugo
authentication unsuccessful

python -m fitz show x.pdf -pass jorjmkie
authenticated as owner
file 'x.pdf', pages: 1, objects: 19, 58 MB, PDF 1.4, encryption: Standard V5 R6 256-
bit AES
Document contains 15 embedded files.

python -m fitz show FDA-1572_508_R6_FINAL.pdf -tr -m
```

(continues on next page)

(continued from previous page)

```
'FDA-1572_508_R6_FINAL.pdf', pages: 2, objects: 1645, 1.4 MB, PDF 1.6, encryption:
↳Standard V4 R4 128-bit AES
document contains 740 root form fields and is signed
```

```
----- PDF metadata -----
format: PDF 1.6
title: FORM FDA 1572
author: PSC Publishing Services
subject: Statement of Investigator
keywords: None
creator: PScript5.dll Version 5.2.2
producer: Acrobat Distiller 9.0.0 (Windows)
creationDate: D:20130522104413-04'00'
modDate: D:20190718154905-07'00'
encryption: Standard V4 R4 128-bit AES

----- PDF trailer -----
<<
/DecodeParms <<
  /Columns 5
  /Predictor 12
>>
/Encrypt 1389 0 R
/Filter /FlateDecode
/ID [ <9252E9E39183F2A0B0C51BE557B8A8FC> <85227BE9B84B724E8F678E1529BA8351> ]
/Index [ 1388 258 ]
/Info 1387 0 R
/Length 253
/Prev 1510559
/Root 1390 0 R
/Size 1646
/Type /XRef
/W [ 1 3 1 ]
>>
```

5.6 Embedded Files Commands

The following commands deal with embedded files – which is a feature completely removed from MuPDF after v1.14, and hence from all its command line tools.

5.6.1 Information

Show the embedded file names (long or short format):

```
python -m fitz embed-info -h
usage: fitz embed-info [-h] [-name NAME] [-detail] [-password PASSWORD] input

----- list embedded files -----

positional arguments:
input                  PDF filename

optional arguments:
```

(continues on next page)

(continued from previous page)

```
-h, --help          show this help message and exit
-name NAME         if given, report only this one
-detail           show detail information
-password PASSWORD password
```

Example:

```
python -m fitz embed-info some.pdf
'some.pdf' contains the following 15 embedded files.

20110813_180956_0002.jpg
20110813_181009_0003.jpg
20110813_181012_0004.jpg
20110813_181131_0005.jpg
20110813_181144_0006.jpg
20110813_181306_0007.jpg
20110813_181307_0008.jpg
20110813_181314_0009.jpg
20110813_181315_0010.jpg
20110813_181324_0011.jpg
20110813_181339_0012.jpg
20110813_181913_0013.jpg
insta-20110813_180944_0001.jpg
markiert-20110813_180944_0001.jpg
neue.datei
```

Detailed output would look like this per entry:

```
name: neue.datei
filename: text-tester.pdf
ufilename: text-tester.pdf
desc: nur zum Testen!
size: 4639
length: 1566
```

5.6.2 Extraction

Extract an embedded file like this:

```
python -m fitz embed-extract -h
usage: fitz embed-extract [-h] -name NAME [-password PASSWORD] [-output OUTPUT]
                           input
----- extract embedded file to disk -----

positional arguments:
input                  PDF filename

optional arguments:
-h, --help          show this help message and exit
-name NAME         name of entry
-password PASSWORD password
-output OUTPUT     output filename, default is stored name
```

For details consult `Document.embeddedFileGet()`. Example (refer to previous section):

```
python -m fitz embed-extract some.pdf -name neue.datei
Saved entry 'neue.datei' as 'text-tester.pdf'
```

5.6.3 Deletion

Delete an embedded file like this:

```
python -m fitz embed-del -h
usage: fitz embed-del [-h] [-password PASSWORD] [-output OUTPUT] -name NAME input

----- delete embedded file -----

positional arguments:
input                  PDF filename

optional arguments:
-h, --help            show this help message and exit
-password PASSWORD    password
-output OUTPUT        output PDF filename, incremental save if none
-name NAME            name of entry to delete
```

For details consult `Document.embeddedFileDel()`.

5.6.4 Insertion

Add a new embedded file using this command:

```
python -m fitz embed-add -h
usage: fitz embed-add [-h] [-password PASSWORD] [-output OUTPUT] -name NAME -path
                    PATH [-desc DESC]
                    input

----- add embedded file -----

positional arguments:
input                  PDF filename

optional arguments:
-h, --help            show this help message and exit
-password PASSWORD    password
-output OUTPUT        output PDF filename, incremental save if none
-name NAME            name of new entry
-path PATH            path to data for new entry
-desc DESC            description of new entry
```

“*NAME*” **must not** already exist in the PDF. For details consult `Document.embeddedFileAdd()`.

5.6.5 Updates

Update an existing embedded file using this command:

```
python -m fitz embed-upd -h
usage: fitz embed-upd [-h] -name NAME [-password PASSWORD] [-output OUTPUT]
                    [-path PATH] [-filename FILENAME] [-filename UFILENAME]
                    [-desc DESC]
                    input

----- update embedded file -----

positional arguments:
input                  PDF filename

optional arguments:
-h, --help            show this help message and exit
-name NAME            name of entry
-password PASSWORD    password
-output OUTPUT        Output PDF filename, incremental save if none
-path PATH            path to new data for entry
-filename FILENAME    new filename to store in entry
-ufilename UFILENAME  new unicode filename to store in entry
-desc DESC            new description to store in entry

except '-name' all parameters are optional
```

Use this method to change meta-information of the file – just omit the “*PATH*”. For details consult *Document.embeddedFileUpd()*.

5.6.6 Copying

Copy embedded files between PDFs:

```
python -m fitz embed-copy -h
usage: fitz embed-copy [-h] [-password PASSWORD] [-output OUTPUT] -source
                    SOURCE [-pwsdsource PWDSOURCE]
                    [-name [NAME [NAME ...]]]
                    input

----- copy embedded files between PDFs -----

positional arguments:
input                  PDF to receive embedded files

optional arguments:
-h, --help            show this help message and exit
-password PASSWORD    password of input
-output OUTPUT        output PDF, incremental save to 'input' if omitted
-source SOURCE        copy embedded files from here
-pwsdsource PWDSOURCE password of 'source' PDF
-name [NAME [NAME ...]]
                    restrict copy to these entries
```

6.1 Annot

This class is supported for PDF documents only.

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

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

Note: Unfortunately, there exists no single, unique naming convention in PyMuPDF: examples for all of *CamelCases*, *mixedCases* and *lower_case_with_underscores* can be found all over the place. We are now in the process of cleaning this up, step by step.

This class, `Annot`, is the first candidate for this exercise. In this chapter, you will for example find `Annot.get_pixmap()` – and no longer the old name `getPixmap`. The method with the old name however **continues to exist** and you can continue using it: your existing code will not break. But we do hope you will start using the new names – for new code at least.

Attribute	Short Description
<code>Annot.delete_responses()</code>	delete all responding annotations
<code>Annot.file_info()</code>	get attached file information
<code>Annot.get_file()</code>	get attached file content
<code>Annot.get_oc()</code>	get <i>xref</i> of an <i>OCG / OCMD</i>
<code>Annot.get_pixmap()</code>	image of the annotation as a pixmap
<code>Annot.get_sound()</code>	get the sound of an audio annotation
<code>Annot.get_text()</code>	extract annotation text
<code>Annot.get_textbox()</code>	extract annotation text

Continued on next page

Table 1 – continued from previous page

Attribute	Short Description
<code>Annot.set_border()</code>	set annotation's border properties
<code>Annot.set_blendmode()</code>	set annotation's blend mode
<code>Annot.set_colors()</code>	set annotation's colors
<code>Annot.set_flags()</code>	set annotation's flags field
<code>Annot.set_name()</code>	set annotation's name field
<code>Annot.set_oc()</code>	set <i>xref</i> to an <i>OCG</i> / <i>OCMD</i>
<code>Annot.set_opacity()</code>	change transparency
<code>Annot.set_open()</code>	open / close annotation or its Popup
<code>Annot.set_popup()</code>	create a Popup for the annotation
<code>Annot.set_rect()</code>	change annotation rectangle
<code>Annot.set_rotation()</code>	change rotation
<code>Annot.update_file()</code>	update attached file content
<code>Annot.update()</code>	apply accumulated annot changes
<code>Annot.blendmode</code>	annotation BlendMode
<code>Annot.border</code>	border details
<code>Annot.colors</code>	border / background and fill colors
<code>Annot.flags</code>	annotation flags
<code>Annot.has_popup</code>	whether annotation has a Popup
<code>Annot.info</code>	various information
<code>Annot.is_open</code>	whether annotation or its Popup is open
<code>Annot.line_ends</code>	start / end appearance of line-type annotations
<code>Annot.next</code>	link to the next annotation
<code>Annot.opacity</code>	the annot's transparency
<code>Annot.parent</code>	page object of the annotation
<code>Annot.popup_rect</code>	rectangle of the annotation's Popup
<code>Annot.popup_xref</code>	the PDF <i>xref</i> number of the annotation's Popup
<code>Annot.rect</code>	rectangle containing the annotation
<code>Annot.type</code>	type of the annotation
<code>Annot.vertices</code>	point coordinates of Polygons, PolyLines, etc.
<code>Annot.xref</code>	the PDF <i>xref</i> number

Class API

class Annot

`get_pixmap` (*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_like*) – 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*

Note: If the annotation has just been created or modified, you should reload the page first via `page =`

doc.reload_page(page).

get_text (*opt="text", clip=None, flags=None*)
(New in 1.18.0)

Retrieves the content of the annotation in a variety of formats – much like the same method for *Page*.. This currently only delivers relevant data for annotation types ‘FreeText’ and ‘Stamp’. Other types return an empty string (or equivalent objects).

Parameters

- **opt** (*str*) – the desired format - one of the following values. Please note that this method works exactly like the same-named method of *Page*.
 - “text” – *TextPage.extractTEXT()*, default
 - “blocks” – *TextPage.extractBLOCKS()*
 - “words” – *TextPage.extractWORDS()*
 - “html” – *TextPage.extractHTML()*
 - “xhtml” – *TextPage.extractXHTML()*
 - “xml” – *TextPage.extractXML()*
 - “dict” – *TextPage.extractDICT()*
 - “json” – *TextPage.extractJSON()*
 - “rawdict” – *TextPage.extractRAWDICT()*
- **clip** (*rect-like*) – restrict the extraction to this area. Should hardly ever be required, defaults to *Annot.rect*.
- **flags** (*int*) – control the amount of data returned. Defaults to simple text extraction.

get_textbox (*rect*)
(New in 1.18.0)

Return the annotation text. Mostly (except line breaks) equal to *Annot.get_text()* with the “text” option.

Parameters *rect* (*rect-like*) – the area to consider, defaults to *Annot.rect*.

set_info (*info=None, content=None, title=None, creationDate=None, modDate=None, subject=None*)
(Changed in version 1.16.10)

Changes annotation properties. These include dates, contents, subject and author (title). Changes for *name* and *id* will be ignored. The update happens selectively: To leave a property unchanged, set it to *None*. To delete existing data, use an empty string.

Parameters

- **info** (*dict*) – a dictionary compatible with the *info* property (see below). All entries must be strings. If this argument is not a dictionary, the other arguments are used instead – else they are ignored.
- **content** (*str*) – (new in v1.16.10) see description in *info*.
- **title** (*str*) – (new in v1.16.10) see description in *info*.
- **creationDate** (*str*) – (new in v1.16.10) date of annot creation. If given, should be in PDF datetime format.

- **modDate** (*str*) – (new in v1.16.10) date of last modification. If given, should be in PDF datetime format.
- **subject** (*str*) – (new in v1.16.10) see description in *info*.

set_line_ends (*start, end*)

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

Note:

- While only 'FreeText', 'Line', 'PolyLine', and 'Polygon' annotations can have these properties, (Py-) MuPDF does not support line ends for 'FreeText', because the call-out variant for these is not supported.
 - (Changed in v1.16.16) Some symbols have an interior area (diamonds, circles, squares, etc.). By default, these areas are filled with the fill color of the annotation. If this is *None*, then white is chosen. The *fill_color* argument of *Annot.update()* can now be used to override this and give line end symbols their own fill color.
-

Parameters

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

set_oc (*xref*)

Set the annotation's visibility using PDF optional content mechanisms. This visibility is controlled by the user interface of supporting PDF viewers. It is independent from other attributes like *Annot.flags*.

Parameters **xref** (*int*) – the *xref* of an optional contents group (OCG or OCMD). Any previous *xref* will be overwritten. If zero, a previous entry will be removed. An exception occurs if the *xref* is not zero and does not point to a valid PDF object.

Note: This does **not require executing** *Annot.update()* to take effect.

get_oc ()

Return the *xref* of an optional content object, or zero if there is none.

Returns zero or the *xref* of an OCG (or OCMD).

set_open (*value*)

(New in v1.18.4)

Set the annotation's Popup annotation to open or closed – **or** the annotation itself, if its type is 'Text' ("sticky note").

Parameters **value** (*bool*) – the desired open state.

set_popup (*rect*)

(New in v1.18.4)

Create a Popup annotation for the annotation and specify its rectangle. If the Popup already exists, only its rectangle is updated.

Parameters **rect** (*rect_like*) – the desired rectangle.

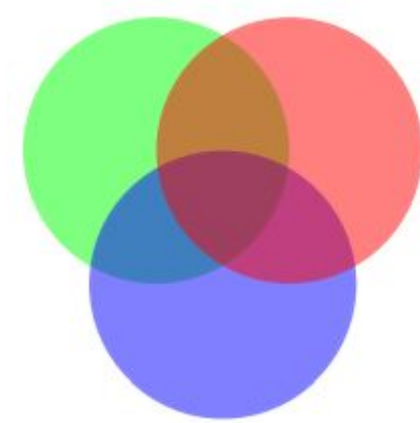
set_opacity (*value*)

Set the annotation's transparency. Opacity can also be set in `Annot.update()`.

Parameters **value** (*float*) – a float in range `[0, 1]`. Any value outside is assumed to be 1.

E.g. a value of 0.5 sets the transparency to 50%.

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

**blendmode**

(New in v1.18.4)

The annotation's blend mode. See *Adobe PDF References*, page 520 for explanations.

Return type `str`

Returns

the blend mode or `None`.

```
>>> annot=page.firstAnnot
>>> annot.blendmode
'Multiply'
```

set_blendmode (*blendmode*)

(New in v1.16.14) Set the annotation's blend mode. See *Adobe PDF References*, page 520 for explanations.

The blend mode can also be set in `Annot.update()`.

Parameters **blendmode** (*str*) – set the blend mode. Use `Annot.update()` to reflect this in the visual appearance. For predefined values see *PDF Standard Blend Modes*. Use `PDF_BM_Normal` to **remove** a blend mode.

```
>>> annot.set_blendmode(fitz.PDF_BM_Multiply)
>>> annot.update()
>>> # or in one statement:
>>> annot.update(blend_mode=fitz.PDF_BM_Multiply, ...)
```

set_name (*name*)

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

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

Caution: If you set the name of a ‘Stamp’ annotation, then this will **not change** the rectangle, nor will the text be layouted in any way. If you choose a standard text from *Stamp Annotation Icons* (the **exact** name piece after “**STAMP_**”), you should receive the original layout. An **arbitrary text** will not be changed to upper case, but be written in font “Times-Bold” as is, horizontally centered in **one line** and be shortened to fit. To get your text fully displayed, its length using fontsize 20 must not exceed 190 pixels. So please make sure that the following inequality is true: `fitz.getTextlength(text, fontname="tibo", fontsize=20) <= 190`.

set_rect (*rect*)

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

Parameters **rect** (*rect_like*) – the new rectangle of the annotation (finite and not empty).

E.g. using a value of `annot.rect + (5, 5, 5, 5)` will shift the annot position 5 pixels to the right and downwards.

Note: You **need not** invoke `Annot.update()` for activation of the effect.

set_rotation (*angle*)

Set the rotation of an annotation. This rotates the annotation rectangle around its center point. Then a **new annotation rectangle** is calculated from the resulting quad.

Parameters **angle** (*int*) – rotation angle in degrees. Arbitrary values are possible, but will be clamped to the interval $0 \leq \text{angle} < 360$.

Note:

- You **must invoke** `Annot.update()` to activate the effect.
 - For PDF_ANNOT_FREE_TEXT, only one of the values 0, 90, 180 and 270 is possible and will **rotate the text** inside the current rectangle (which remains unchanged). Other values are silently ignored and replaced by 0.
 - Otherwise, only the following *Annotation Types* can be rotated: ‘Square’, ‘Circle’, ‘Caret’, ‘Text’, ‘FileAttachment’, ‘Ink’, ‘Line’, ‘Polyline’, ‘Polygon’, and ‘Stamp’. For all others the method is a no-op.
-

set_border (*border=None, width=0, style=None, dashes=None*)

PDF only: Change border width and dashing properties.

Changed in version 1.16.9: Allow specification without using a dictionary. The direct parameters are used if *border* is not a dictionary.

Parameters

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

set_flags (*flags*)

Changes the annotation flags. Use the `|` operator to combine several.

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

set_colors (*colors=None, stroke=None, fill=None*)

Changes the “stroke” and “fill” colors for supported annotation types – not all annotations accept both.

Changed in version 1.16.9: Allow colors to be directly set. These parameters are used if *colors* is not a dictionary.

Parameters

- **colors** (*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.
- **stroke** (*sequence*) – see above.
- **fill** (*sequence*) – see above.

Changed in v1.18.5: To completely remove a color specification, use an empty sequence like `[]`.

delete_responses ()

(New in version 1.16.12) Delete annotations referring to this one. This includes any ‘Popup’ annotations and all annotations responding to it.

update (*opacity=None, blend_mode=None, fontsize=0, text_color=None, border_color=None, fill_color=None, cross_out=True, rotate=-1*)

Synchronize the appearance of an annotation with its properties after any changes.

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

- `set_rect()`
- `set_flags()`
- `set_oc()`
- `update_file()`
- `set_info()` (except any changes to “content”)

All arguments are optional. *(Changed in v1.16.14)* Blend mode and opacity are applicable to **all annotation types**. The other arguments are mostly special use, as described below.

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

Parameters

- **opacity** (*float*) – *(new in v1.16.14)* **valid for all annotation types:** change or set the annotation’s transparency. Valid values are $0 \leq \text{opacity} < 1$.
- **blend_mode** (*str*) – *(new in v1.16.14)* **valid for all annotation types:** change or set the annotation’s blend mode. For valid values see [PDF Standard Blend Modes](#).
- **fontsize** (*float*) – change font size of the text. ‘FreeText’ annotations only.
- **text_color** (*sequence, float*) – change the text color. ‘FreeText’ annotations only.
- **border_color** (*sequence, float*) – change the border color. ‘FreeText’ annotations only.

- **fill_color** (*sequence, float*) – the fill color.
 - ‘Line’, ‘Polyline’, ‘Polygon’ annotations: use it to give applicable line end symbols a fill color other than that of the annotation (*changed in v1.16.16*).
- **cross_out** (*bool*) – (*new in v1.17.2*) add two diagonal lines to the annotation rectangle. ‘Redact’ annotations only. If not desired, *False* must be specified even if the annotation was created with *False*.
- **rotate** (*int*) – new rotation value. Default (-1) means no change. Supports ‘FreeText’ and several other annotation types (see `Annot.setRotation()`),¹. Only choose 0, 90, 180, or 270 degrees for ‘FreeText’. Otherwise any integer is acceptable.

Return type bool

file_info ()

Basic information of the annot’s attached file.

Return type dict

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

get_file ()

Returns attached file content.

Return type bytes

Returns the content of the attached file.

update_file (*buffer=None, filename=None, ufilename=None, desc=None*)

Updates the content of an attached file. All arguments are optional. No arguments lead to a no-op.

Parameters

- **buffer** (*bytes|bytearray|BytesIO*) – the new file content. Omit to only change meta-information.
(Changed in version 1.14.13) io.BytesIO is now also supported.
- **filename** (*str*) – new filename to associate with the file.
- **ufilename** (*str*) – new unicode filename to associate with the file.
- **desc** (*str*) – new description of the file content.

get_sound ()

Return the embedded sound of an audio annotation.

Return type dict

Returns

the sound audio file and accompanying properties. These are the possible dictionary keys, of which only “rate” and “stream” are always present.

¹ Rotating an annotation generally also changes its rectangle. Depending on how the annotation was defined, the original rectangle in general is **not reconstructible** by setting the rotation value to zero. This information may be lost.

Key	Description
rate	(float, requ.) samples per second
channels	(int, opt.) number of sound channels
bps	(int, opt.) bits per sample value per channel
encoding	(str, opt.) encoding format: Raw, Signed, muLaw, ALaw
compression	(str, opt.) name of compression filter
stream	(bytes, requ.) the sound file content

opacity

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

Return type float

parent

The owning page object of the annotation.

Return type *Page*

rotation

The annot rotation.

Return type int

Returns a value $[-1, 359]$. If rotation is not at all, -1 is returned (and implies a rotation angle of 0). Other possible values are normalized to some value $0 \leq \text{angle} < 360$.

rect

The rectangle containing the annotation.

Return type *Rect*

next

The next annotation on this page or None.

Return type *Annot*

type

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

Return type list

info

A dictionary containing various information. All fields are optional strings. If an information is not provided, an empty string is returned.

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

- *id* – (new in version 1.16.10) a unique identification of the annotation. This is taken from PDF key */NM*. Annotations added by PyMuPDF will have a unique name, which appears here.

Return type dict

flags

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

Return type int

line_ends

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

Return type tuple

vertices

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

- ‘Line’ – the starting and ending coordinates (2 float pairs).
- ‘FreeText’ – 2 or 3 float pairs designating the starting, the (optional) knee point, and the ending coordinates.
- ‘PolyLine’ / ‘Polygon’ – the coordinates of the edges connected by line pieces (n float pairs for n points).
- text markup annotations – 4 float pairs specifying the *QuadPoints* of the marked text span (see *Adobe PDF References*, page 634).
- ‘Ink’ – list of one to many sublists of vertex coordinates. Each such sublist represents a separate line in the drawing.

Return type list

colors

dictionary of two lists of floats in range $0 \leq float \leq 1$ specifying the “stroke” and the interior (“fill”) colors. The stroke color is used for borders and everything that is actively painted or written (“stroked”). The fill color is used for the interior of objects like line ends, circles and squares. The lengths of these lists implicitly determine the colorspaces used: 1 = GRAY, 3 = RGB, 4 = CMYK. So “[1.0, 0.0, 0.0]” stands for RGB color red. Both lists can be empty if no color is specified.

Return type dict

xref

The PDF *xref*.

Return type int

popup_xref

The PDF *xref* of the associated Popup annotation. Zero if non-existent.

Return type int

has_popup

Whether the annotation has a Popup annotation.

Return type bool

is_open

Whether the annotation's Popup is open – **or** the annotation itself ('Text' annotations only).

Return type bool

popup_rect

The rectangle of the associated Popup annotation. Infinite rectangle if non-existent.

Return type *Rect*

border

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

- *width* – a float indicating the border thickness in points. The value is -1.0 if no width is specified.
- *dashes* – a sequence of integers specifying a line dash pattern. *[]* means no dashes, *[n]* means equal on-off lengths of *n* points, longer lists will be interpreted as specifying alternating on-off length values. See the *Adobe PDF References* 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

6.1.1 Annotation Icons in MuPDF

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

-  PushPin
-  Graph
-  Paperclip
-  Tag
-  Note
-  Comment
-  Help
-  Insert
-  Key
-  NewParagraph
-  Paragraph
-  Mic
-  Speaker
-  Star (default if none of the above)

6.1.2 Example

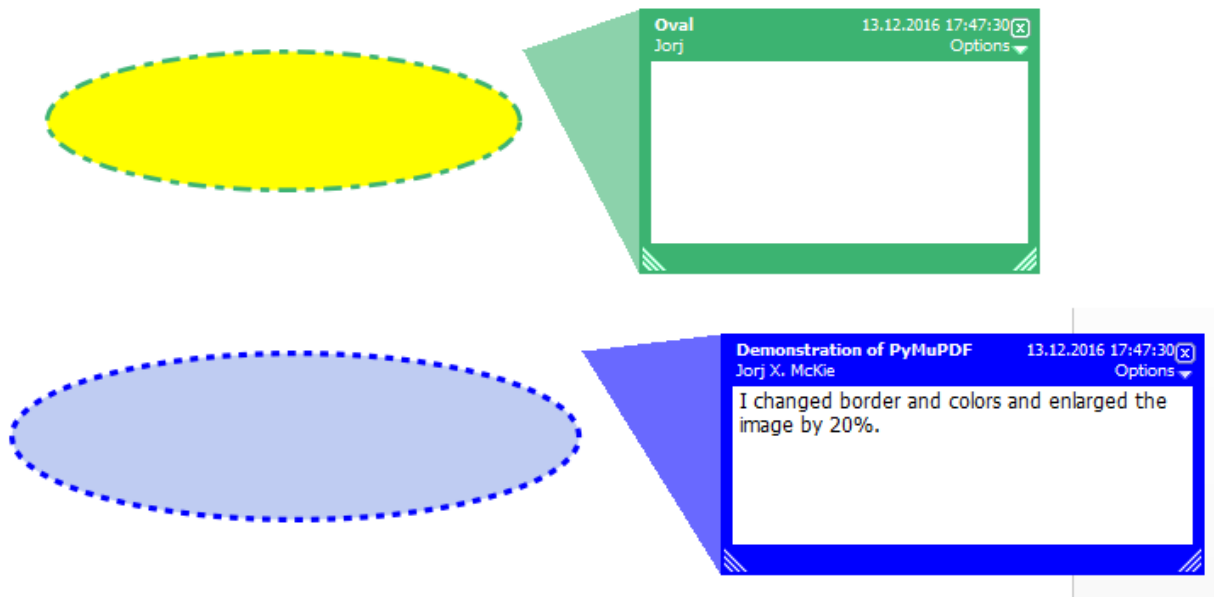
Change the graphical image of an annotation. Also update the “author” and the text to be shown in the popup window:

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

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

# text in popup window ...
info["content"] = "I changed border and colors and enlarged the image by 20%."
info["subject"] = "Demonstration of PyMuPDF"  # some PDF viewers also show this
annot.set_info(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.set_rect(r)                             # update rectangle
annot.update()                               # update the annot's appearance
doc.save("circle-out.pdf")                    # save
```

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



6.2 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)*

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 display list 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
<code>run()</code>	Run a display list through a device.
<code>getPixmap()</code>	generate a pixmap
<code>getTextPage()</code>	generate a text page
<code>rect</code>	mediabox of the display list

Class API

class DisplayList

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

Parameters `mediabox` (*Rect*) – The page’s rectangle.

Return type *DisplayList*

`run(device, matrix, area)`

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

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

Parameters

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

`getPixmap(matrix=fitz.Identity, colorspace=fitz.csRGB, alpha=0, clip=None)`

Run the display list through a draw device and return a pixmap.

Parameters

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

Return type *Pixmap*

Returns pixmap of the display list.

getTextPage (*flags*)

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

Parameters flags (*int*) – control which information is parsed into a text page. Default value in PyMuPDF is `3 = TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE`, i.e. ligatures are **passed through**, 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 Document

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

There exists the alias *open* for this class, i.e. `fitz.Document(...)` and `fitz.open(...)` do exactly the same thing.

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

Note: Starting with v1.17.0, a new page addressing mechanism for **EPUB files only** is supported. This document type is internally organized in chapters such that pages can most efficiently be found by their so-called “location”. The location is a tuple (*chapter; pno*) consisting of the chapter number and the page number **in that chapter**. Both numbers are zero-based.

While it is still possible to locate a page via its (absolute) number, doing so may mean that the complete document has to be layouted before the page can be addressed. This may have a significant performance implication if the document is very large. Due to internal EPUB file structures, using the page’s **location** (*chapter; pno*) prevents this from happening.

To maintain a consistent API, PyMuPDF supports page *location* syntax for **all file types** – documents without this feature simply have just one chapter. *Document.loadPage()* and the equivalent index access now also support using the page *location*. There are a number of methods to convert between page numbers and locations, determine the chapter count, the page count per chapter, to compute the next and previous locations, and the last page location of a document.

Method / Attribute	Short Description
<code>Document.add_layer_config()</code>	PDF only: make new optional content configuration
<code>Document.add_ocg()</code>	PDF only: add new optional content group
<code>Document.authenticate()</code>	gain access to an encrypted document
<code>Document.can_save_incrementally()</code>	check if incremental save is possible
<code>Document.chapterPageCount()</code>	number of pages in chapter
<code>Document.close()</code>	close the document

Continued on next page

Table 2 – continued from previous page

Method / Attribute	Short Description
<code>Document.convertToPDF()</code>	write a PDF version to memory
<code>Document.copyPage()</code>	PDF only: copy a page reference
<code>Document.del_toc_item()</code>	PDF only: remove a single TOC item
<code>Document.deletePage()</code>	PDF only: delete a page
<code>Document.deletePageRange()</code>	PDF only: delete a page range
<code>Document.embeddedFileAdd()</code>	PDF only: add a new embedded file from buffer
<code>Document.embeddedFileCount()</code>	PDF only: number of embedded files
<code>Document.embeddedFileDel()</code>	PDF only: delete an embedded file entry
<code>Document.embeddedFileGet()</code>	PDF only: extract an embedded file buffer
<code>Document.embeddedFileInfo()</code>	PDF only: metadata of an embedded file
<code>Document.embeddedFileNames()</code>	PDF only: list of embedded files
<code>Document.embeddedFileUpd()</code>	PDF only: change an embedded file
<code>Document.findBookmark()</code>	retrieve page location after layouting
<code>Document.fullcopyPage()</code>	PDF only: duplicate a page
<code>Document.get_oc()</code>	PDF only: get OCG /OCMD xref of image / form xobject
<code>Document.get_oc_states()</code>	PDF only: lists of OCGs in ON, OFF, RBGroups
<code>Document.get_ocgs()</code>	PDF only: info on all optional content groups
<code>Document.get_ocmd()</code>	PDF only: retrieve definition of an <i>OCMD</i>
<code>Document.get_toc()</code>	alias of <code>getToC()</code>
<code>Document.getPageFontList()</code>	PDF only: make a list of fonts on a page
<code>Document.getPageImageList()</code>	PDF only: make a list of images on a page
<code>Document.getPagePixmap()</code>	create a pixmap of a page by page number
<code>Document.getPageText()</code>	extract the text of a page by page number
<code>Document.getPageXObjectList()</code>	PDF only: make a list of XObjects on a page
<code>Document.getSigFlags()</code>	PDF only: determine signature state
<code>Document.getToC()</code>	create a table of contents
<code>Document.getXmlMetadata()</code>	PDF only: read the XML metadata
<code>Document.insertPage()</code>	PDF only: insert a new page
<code>Document.insertPDF()</code>	PDF only: insert pages from another PDF
<code>Document.layer_configs()</code>	PDF only: list of optional content configurations
<code>Document.layer_ui_configs()</code>	PDF only: list of optional content intents
<code>Document.layout()</code>	re-paginate the document (if supported)
<code>Document.loadPage()</code>	read a page
<code>Document.makeBookmark()</code>	create a page pointer in reflowable documents
<code>Document.metadataXML()</code>	PDF only: <i>xref</i> of XML metadata
<code>Document.movePage()</code>	PDF only: move a page to different location in doc
<code>Document.need_appearances()</code>	PDF only: <code>get/set /NeedAppearances</code> property
<code>Document.newPage()</code>	PDF only: insert a new empty page
<code>Document.nextLocation()</code>	return (chapter, pno) of following page
<code>Document.outlineXref()</code>	PDF only: <i>xref</i> a TOC item
<code>Document.pageCropBox()</code>	PDF only: the unrotated page rectangle
<code>Document.pages()</code>	iterator over a page range
<code>Document.pageXref()</code>	PDF only: <i>xref</i> of the page
<code>Document.PDFCatalog()</code>	PDF only: <i>xref</i> of catalog (root)
<code>Document.PDFTrailer()</code>	PDF only: trailer source
<code>Document.previousLocation()</code>	return (chapter, pno) of preceding page
<code>Document.reload_page()</code>	PDF only: provide a new copy of a page
<code>Document.save()</code>	PDF only: save the document
<code>Document.saveIncr()</code>	PDF only: save the document incrementally

Continued on next page

Table 2 – continued from previous page

Method / Attribute	Short Description
<code>Document.scrub()</code>	PDF only: remove sensitive data
<code>Document.searchPageFor()</code>	search for a string on a page
<code>Document.select()</code>	PDF only: select a subset of pages
<code>Document.set_oc()</code>	PDF only: attach OCG / OCMD to image / form xobject
<code>Document.set_oc_states()</code>	PDF only: mass changing OCG states
<code>Document.set_occmd()</code>	PDF only: create or update an <i>OCMD</i>
<code>Document.set_toc_item()</code>	PDF only: change a single TOC item
<code>Document.set_toc()</code>	PDF only: alias of <code>setToC</code>
<code>Document.set_layer_ui_config()</code>	PDF only: temporarily set the visibility of OCGs
<code>Document.setMetadata()</code>	PDF only: set the metadata
<code>Document.setToC()</code>	PDF only: set the table of contents (TOC)
<code>Document.setXmlMetadata()</code>	PDF only: create or update document XML metadata
<code>Document.updateObject()</code>	PDF only: replace object source
<code>Document.updateStream()</code>	PDF only: replace stream source
<code>Document.write()</code>	PDF only: writes document to memory
<code>Document.xrefObject()</code>	PDF only: object source at the <i>xref</i>
<code>Document.xrefStream()</code>	PDF only: expanded stream source at <i>xref</i>
<code>Document.xrefStreamRaw()</code>	PDF only: raw stream source at <i>xref</i>
<code>Document.chapterCount</code>	number of chapters
<code>Document.FormFonts</code>	PDF only: list of global widget fonts
<code>Document.isClosed</code>	has document been closed?
<code>Document.isDirty</code>	PDF only: has document been changed yet?
<code>Document.isEncrypted</code>	document (still) encrypted?
<code>Document.isFormPDF</code>	is this a Form PDF?
<code>Document.isPDF</code>	is this a PDF?
<code>Document.isReflowable</code>	is this a reflowable document?
<code>Document.isRepaired</code>	PDF only: has this PDF been repaired during open?
<code>Document.lastLocation</code>	(chapter, pno) of last page
<code>Document.metadata</code>	metadata
<code>Document.name</code>	filename of document
<code>Document.needsPass</code>	require password to access data?
<code>Document.outline</code>	first <i>Outline</i> item
<code>Document.pageCount</code>	number of pages
<code>Document.permissions</code>	permissions to access the document

Class API

class Document

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

Creates a *Document* object.

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

Parameters

- **filename** (*str, pathlib*) – A UTF-8 string or *pathlib* object containing a file path (or a file type, see below).
- **stream** (*bytes, bytearray, BytesIO*) – A memory area containing a supported document. Its type **must** be specified by either *filename* or *filetype*.
(*Changed in version 1.14.13*) *io.BytesIO* is now also supported.
- **filetype** (*str*) – A string specifying the type of document. This may be something looking like a filename (e.g. “x.pdf”), in which case MuPDF uses the extension to determine the type, or a mime type like *application/pdf*. Just using strings like “pdf” will also work.
- **rect** (*rect_like*) – a rectangle specifying the desired page size. This parameter is only meaningful for documents with a variable page layout (“reflowable” documents), like e-books or HTML, and ignored otherwise. If specified, it must be a non-empty, finite rectangle with top-left coordinates (0, 0). Together with parameter *fontsize*, each page will be accordingly laid out and hence also determine the number of pages.
- **width** (*float*) – may be used together with *height* as an alternative to *rect* to specify layout information.
- **height** (*float*) – may be used together with *width* as an alternative to *rect* to specify layout information.
- **fontsize** (*float*) – the default fontsize for reflowable document types. This parameter is ignored if none of the parameters *rect* or *width* and *height* are specified. Will be used to calculate the page layout.

Overview of possible forms (*open* is a synonym of *Document*):

```
>>> # from a file
>>> doc = fitz.open("some.pdf")
>>> doc = fitz.open("some.file", None, "pdf") # copes with wrong extension
>>> doc = fitz.open("some.file", filetype="pdf") # copes with wrong extension
>>>
>>> # from memory
>>> doc = fitz.open("pdf", mem_area)
>>> doc = fitz.open(None, mem_area, "pdf")
>>> doc = fitz.open(stream=mem_area, filetype="pdf")
>>>
>>> # new empty PDF
>>> doc = fitz.open()
>>>
```

The Document class can be also be used as a **context manager**. On exit, the document will automatically be closed.

```
>>> import fitz
>>> with fitz.open(...) as doc:
    for page in doc: print("page %i" % page.number)
page 0
page 1
page 2
page 3
>>> doc.isClosed
True
>>>
```


get_oc (*xref*)
(New in v1.18.4)

Return the cross reference number of an *OCG* or *OCMD* attached to an image or form xobject.

Parameters *xref* (*int*) – the *xref* of an image or form xobject. Valid such cross reference numbers are returned by `Document.getPageImageList()`, resp. `Document.getPageXObjectList()`. For invalid numbers, an exception is raised.

Return type *int*

Returns the cross reference number of an optional contents object or zero if there is none.

set_oc (*xref*, *ocxref*)
(New in v1.18.4)

If *xref* represents an image or form xobject, set or remove the cross reference number *ocxref* of an optional contents object.

Parameters

- **xref** (*int*) – the *xref* of an image or form xobject⁵. Valid such cross reference numbers are returned by `Document.getPageImageList()`, resp. `Document.getPageXObjectList()`. For invalid numbers, an exception is raised.
- **ocxref** (*int*) – the *xref* number of an *OCG* / *OCMD*. If not zero, an invalid reference raises an exception. If zero, any OC reference is removed.

layer_configs ()
(New in v1.18.3)

Show optional layer configurations. There always is a standard one, which is not included in the response.

```
>>> for item in doc.layer_configs: print(item)
{'number': 0, 'name': 'my-config', 'creator': ''}
>>> # use 'number' as config identifier in add_ocg
```

add_layer_config (*name*, *creator=None*, *on=None*)
(New in v1.18.3)

Add an optional content configuration. Layers serve as a collection of ON / OFF states for optional content groups. They allow fast visibility switches between different views on the same document.

Parameters

- **name** (*str*) – arbitrary name.
- **creator** (*str*) – creating software.
- **on** (*sequ*) – a sequence of *OCG xref* numbers which should be set to ON (visible). All other *OCGs* will be set to OFF.

set-layer-config (*number*, *as_default=False*)
(New in v1.18.3)

Switch to a document view as defined by the optional layer's configuration number. This is temporary, except if established as default.

Parameters

- **number** (*int*) – config number as returned by `Document.layer_configs()`.
- **as_default** (*bool*) – make this the default configuration.

⁵ Examples for "Form XObjects" are created by `Page.showPDFpage()`.

Activates the ON / OFF states of OCGs as defined in this layer. If *as_default=True*, then additionally all layers, including the standard one, are merged and the result is written back to the standard layer, and **all optional layers are deleted**.

add_ocg (*name*, *config=-1*, *on=True*, *intent="View"*, *usage="Artwork"*)
(New in v1.18.3)

Add an optional content group. An OCG is the most important unit of information to determine object visibility. For a PDF, in order to be regarded as having optional content, at least one OCG must exist.

Parameters

- **name** (*str*) – arbitrary name. Will show up in supporting PDF viewers.
- **config** (*int*) – layer configuration number. Default -1 is the standard configuration.
- **on** (*bool*) – standard visibility status for objects pointing to this OCG.
- **intent** (*str*, *list*) – a string or list of strings declaring the visibility intents. There are two PDF standard values to choose from: “View” and “Design”. Default is “View”. **Correct spelling is important.**
- **usage** (*str*) – another influencer for OCG visibility. This will become part of the OCG’s /Usage key. There are two PDF standard values to choose from: “Artwork” and “Technical”. Default is “Artwork”. Please only change when required.

Returns *xref* of the created OCG. Use as entry for *oc* parameter in supporting objects.

Note: Multiple OCGs with identical parameters may be created. This will not cause problems. Garbage option 3 of *Document.save()* will get rid of any duplicates.

set_ocmd (*xref=0*, *ocgs=None*, *policy="AnyOn"*, *ve=None*)
(New in v1.18.4)

Create or update an *OCMD* (optional content membership dictionary).

Parameters

- **xref** (*int*) – *xref* of the OCMD to be updated, or 0 for a new OCMD.
- **ocgs** (*list*) – a sequence of *xref* numbers of existing *OCG* PDF objects.
- **policy** (*str*) – one of “AnyOn” (default), “AnyOff”, “AllOn”, “AllOff” (mixed or lower case).
- **ve** (*list*) – a “visibility expression”. This is a list of arbitrarily nested other lists – see explanation below. Use as an alternative to the combination *ocgs* / *policy* if you need to formulate more complex conditions.

Return type *int*

Returns *xref* of the OCMD. Use as *oc=xref* parameter in supporting objects, and respectively in *Document.set_oc()* or *Annot.set_oc()*.

Note: The purpose of OCMDs is to more flexibly determine visibility. An OCMD actually is a boolean expression: it evaluates the current visibility of one or more optional content groups and then computes its own ON (true) or OFF (false) state.

There are two ways to formulate the OCMD’s visibility:

1. Use the combination of *ocgs* and *policy*: The *policy* value is interpreted as follows:

- AnyOn – (default) true if at least one OCG is ON.
- AnyOff – true if at least one OCG is OFF.
- AllOn – true if all OCGs are ON.
- AllOff – true if all OCGs are OFF.

Suppose you want two PDF objects be displayed exactly one at a time (if one is ON, then the other one must be OFF):

Solution: use an **OCG** for object 1 and an **OCMD** for object 2. Create the OCMD via `set_ocmd(ocgs=[xref], policy="AllOff")`, with the *xref* of the OCG.

2. Use the **visibility expression** *ve*: This is a list of a logical expression keyword (string) followed by integers or other lists. The possible logical expressions are “**and**”, “**or**”, and “**not**”. The integers must be *xref* numbers of OCGs. The syntax of this parameter is a bit awkward, but quite powerful:

- Each list, including the top one, must start with a logical expression.
- If the first item is a “**not**”, then the list must have exactly two items. If it is “**and**” or “**or**”, any number of other items may follow.
- Items following the logical expression may be either integers or other lists. An *integer* must be the *xref* of an OCG. A *list* must conform to the rules above.

Examples:

- `set_ocmd(ve=["or", 4, ["not", 5], ["and", 6, 7]])`. This delivers ON if the following is true: “**4 is ON, or 5 is OFF, or 6 and 7 are both ON**”.
- `set_ocmd(ve=["not", xref])`. This has the same effect as the OCMD example created under 1.

For more details and examples see page 367 of *Adobe PDF References*. Also do have a look at example scripts [here](#).

Visibility expressions, `/VE`, are part of the PDF version 1.6 specification. If you are using an older PDF consumer software, you hence may find it unsupported (i.e. ignored).

`get_ocmd(xref)`
(New in v1.18.4)

Retrieve the definition of an OCMD (optional content membership dictionary).

Parameters `xref` (*int*) – the *xref* of the OCMD.

Return type dict

Returns a dictionary with the keys *xref*, *ocgs*, *policy* and *ve*.

`get_oc_states(config=-1)`
(New in v1.18.3)

List of optional content groups by status in the specified configuration. This is a dictionary with lists of cross reference numbers for OCGs that occur in the arrays `/ON`, `/OFF` or in some radio button group (`/RBGroups`).

Parameters `config` (*int*) – the configuration layer (default is the standard config layer).

```
>>> pprint(doc.get_oc_states())
{'off': [8, 9, 10], 'on': [5, 6, 7], 'rbgroups': [[7, 10]]}
>>>
```

set_oc_states (*config*, *on=None*, *off=None*, *basestate=None*, *rbgroups=None*)
(New in v1.18.3)

Mass status changes of optional content groups. **Permanently** sets the status of OCGs.

Parameters

- **config** (*int*) – desired configuration layer, choose -1 for the default one.
- **on** (*list*) – list of *xref* of OCGs to set ON. Replaces previous values. An empty list will cause no OCG being set to ON anymore. Should be specified if *basestate="ON"* is used.
- **off** (*list*) – list of *xref* of OCGs to set OFF. Replaces previous values. An empty list will cause no OCG being set to OFF anymore. Should be specified if *basestate="OFF"* is used.
- **basestate** (*str*) – desired state of OCGs that are not mentioned in *on* resp. *off*. Possible values are “ON”, “OFF” or “Unchanged”. Upper / lower case possible.
- **rbgroups** (*list*) – a list of lists. Replaces previous values. Each sublist should contain two or more OCG xrefs. OCGs in the same sublist are handled like buttons in a radio button group: setting one to ON automatically sets all other group members to OFF.

Values *None* will not change the corresponding PDF array.

```
>>> doc.set_oc_states(-1, basestate="OFF") # only changes the base state
>>> pprint(doc.get_oc_states())
{'basestate': 'OFF', 'off': [8, 9, 10], 'on': [5, 6, 7], 'rbgroups': [[7, 10]]}
```

get_ocgs ()
(New in v1.18.3)

Details of all optional content groups. This is a dictionary of dictionaries like this (key is the OCG’s *xref*):

```
>>> pprint(doc.get_ocgs())
{13: {'on': True,
      'intent': ['View', 'Design'],
      'name': 'Circle',
      'usage': 'Artwork'},
 14: {'on': True,
      'intent': ['View', 'Design'],
      'name': 'Square',
      'usage': 'Artwork'},
 15: {'on': False, 'intent': ['View'], 'name': 'Square', 'usage': 'Artwork'}}
>>>
```

layer_ui_configs ()
(New in v1.18.3)

Show the visibility status of optional content that is modifiable by the user interface of supporting PDF viewers. Example:

```
>>> pprint(doc.layer_ui_configs())
({'depth': 0,
  'locked': False,
  'number': 0,
  'on': True,
  'text': 'Circle',
  'type': 'checkbox'},
 {'depth': 0,
  'locked': False,
  'number': 1,
  'on': False,
  'text': 'Square',
  'type': 'checkbox'})
>>> # refers to OCGs named "Circle" (ON), resp. "Square" (OFF)
```

Note:

- Only reports items contained in the currently selected layer configuration.
- **The meaning of the dictionary keys is as follows:**
 - *depth*: item’s nesting level in the */Order* array
 - *locked*: whether changing the item’s state is prohibited
 - *number*: running sequence number
 - *on*: item state
 - *text*: text string or name field of the originating OCG
 - *type*: one of “label” (set by a text string), “checkbox” (set by a single OCG) or “checkbox” (set by a set of connected OCGs)

set_layer_ui_config (*number*, *action=0*)

(New in v1.18.3)

Modify OC visibility status of content groups. This is analog to what supporting PDF viewers would offer.

Note: Visibility is **not** a property stored with the OCG. It is not even an information necessarily present in the PDF document at all. Instead, the current visibility is **temporarily** set using the user interface of some supporting PDF consumer software. The same type of functionality is offered by this method.

To make **permanent** changes, use `Document.set_oc_states()`.

Parameters

- **number** (*in*) – number as returned by `Document.layer_ui_configs()`.
- **action** (*int*) – 0 = set on (default), 1 = toggle on/off, 2 = set off.

Example:

```
>>> # let's make above "Square" visible:
>>> doc.set_layer_ui_config(1, action=0)
>>> pprint(doc.layer_ui_configs())
({'depth': 0,
```

(continues on next page)

(continued from previous page)

```
'locked': False,
'number': 0,
'on': True,
'text': 'Circle',
'type': 'checkbox'},
{'depth': 0,
'locked': False,
'number': 1,
'on': True, # <===
'text': 'Square',
'type': 'checkbox'})
>>>
```

authenticate (*password*)

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

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

Return type *int*

Returns

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

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

makeBookmark (*loc*)

(New in v.1.17.3) Return a page pointer in a reflowable document. After re-laying out the document, the result of this method can be used to find the new location of the page.

Note: Do not confuse with items of a table of contents, TOC.

Parameters *loc* (*list, tuple*) – page location. Must be a valid (*chapter, pno*).

Return type *pointer*

Returns a long integer in pointer format. To be used for finding the new location of the page after re-laying out the document. Do not touch or re-assign.

findBookmark (*bookmark*)

(New in v.1.17.3) Return the new page location after re-laying out the document.

Parameters *bookmark* (*pointer*) – created by *Document.makeBookmark()*.

Return type *tuple*

Returns the new (*chapter, pno*) of the page.

chapterPageCount (*chapter*)

(New in v.1.17.0) Return the number of pages of a chapter.

Parameters `chapter` (*int*) – the 0-based chapter number.

Return type `int`

Returns number of pages in chapter. Relevant only for document types which support chapter support (EPUB currently).

nextLocation (*page_id*)

(*New in v.1.17.0*) Return the location of the following page.

Parameters `page_id` (*tuple*) – the current page id. This must be a tuple (*chapter*; *pno*) identifying an existing page.

Returns The tuple of the following page, i.e. either (*chapter*; *pno + 1*) or (*chapter + 1*, 0), or the empty tuple () if the argument was the last page. Relevant only for document types which support chapter support (EPUB currently).

previousLocation (*page_id*)

(*New in v.1.17.0*) Return the locator of the preceding page.

Parameters `page_id` (*tuple*) – the current page id. This must be a tuple (*chapter*; *pno*) identifying an existing page.

Returns The tuple of the preceding page, i.e. either (*chapter*; *pno - 1*) or the last page of the preceding chapter, or the empty tuple () if the argument was the first page. Relevant only for document types which support chapter support (EPUB currently).

loadPage (*page_id=0*)

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

(*Changed in v1.17.0*) For document types supporting a so-called “chapter structure” (like EPUB), pages can also be loaded via the combination of chapter number and relative page number, instead of the absolute page number. This should **significantly speed up access** for large documents.

Parameters `page_id` (*int*, *tuple*) – (*Changed in v1.17.0*)

Either a 0-based page number, or a tuple (*chapter*; *pno*). For an **integer**, any $-inf < page_id < pageCount$ is acceptable. While `page_id` is negative, `pageCount` will be added to it. For example: to load the last page, you can use `doc.loadPage(-1)`. After this you have `page.number = doc.pageCount - 1`.

For a tuple, *chapter* must be in range `Document.chapterCount`, and *pno* must be in range `Document.chapterPageCount()` of that chapter. Both values are 0-based. Using this notation, `Page.number` will equal the given tuple. Relevant only for document types which support chapter support (EPUB currently).

Return type *Page*

Note: Documents also follow the Python sequence protocol with page numbers as indices: `doc.loadPage(n) == doc[n]`.

For **absolute page numbers** only, expressions like “*for page in doc: ...*” and “*for page in reversed(doc): ...*” will successively yield the document’s pages. Refer to `Document.pages()` which allows processing pages as with slicing.

You can also use index notation with the new chapter-based page identification: use `page = doc[(5, 2)]` to load the third page of the sixth chapter.

To maintain a consistent API, for document types not supporting a chapter structure (like PDFs), `Document.`

`chapterCount` is 1, and pages can also be loaded via tuples $(0, pno)$. See this³ footnote for comments on performance improvements.

reload_page (*page*)

(New in version 1.16.10)

PDF only: Provide a new copy of a page after finishing and updating all pending changes.

Parameters `page` (*Page*) – page object.

Return type *Page*

Returns a new copy of the same page. All pending updates (e.g. to annotations or widgets) will be finalized and a fresh copy of the page will be loaded. .. note:: In a typical use case, a page *Pixmap* should be taken after annotations / widgets have been added or changed. To force all those changes being reflected in the page structure, this method re-instates a fresh copy while keeping the object hierarchy “document -> page -> annotation(s)” intact.

pageCropBox (*pno*)

(New in version 1.17.7)

PDF only: Return the unrotated page rectangle – **without reading the page (via *Document.loadPage()*). This is meant for internal purpose requiring best possible performance.

Parameters `pno` (*int*) – 0-based page number.

Returns *Rect* of the page like *Page.rect()*, but ignoring any rotation.

pageXref (*pno*)

(New in version 1.17.7)

PDF only: Return the *xref* of the page – **without reading the page (via *Document.loadPage()*). This is meant for internal purpose requiring best possible performance.

Parameters `pno` (*int*) – 0-based page number.

Returns *xref* of the page like *Page.xref*.

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

(New in version 1.16.4)

A generator for a given range of pages. Parameters have the same meaning as in the built-in function *range()*. Intended for expressions of the form “for page in doc.pages(*start*, *stop*, *step*): ...”.

Parameters

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

³ For applicable (EPUB) document types, loading a page via its absolute number may result in layouting a large part of the document, before the page can be accessed. To avoid this performance impact, prefer chapter-based access. Use convenience methods / attributes *Document.nextLocation()*, *Document.previousLocation()* and *Document.lastLocation* for maintaining a high level of coding efficiency.

Returns

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

- `"doc.pages()"` emits all pages.
- `"doc.pages(4, 9, 2)"` emits pages 4, 6, 8.
- `"doc.pages(0, None, 2)"` emits all pages with even numbers.
- `"doc.pages(-2)"` emits the last two pages.
- `"doc.pages(-1, -1)"` emits all pages in reversed order.
- `"doc.pages(-1, -10)"` emits pages in reversed order, starting with the last page **repeatedly**. For a 4-page document the following page numbers are emitted: 3, 2, 1, 0, 3, 2, 1, 0, 3, 2, 1, 0, 3.

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

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

Parameters

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

Return type bytes

Returns

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

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

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

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

- Image files: perfect, no issues detected. Apparently however, image transparency is ignored. If you need that (like for a watermark), use `Page.insertImage()` instead. Otherwise, this method is recommended for its much better performance.
 - XPS: appearance very good. Links work fine, outlines (bookmarks) are lost, but can easily be recovered².
 - EPUB, CBZ, FB2: similar to XPS.
 - SVG: medium. Roughly comparable to `svglib`.
-

`getToC (simple=True)`

`getTOC (simple=True)`

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

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

Return type `list`

Returns

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

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

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

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

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

Return type `Pixmap`

`getPageXObjectList (pno)`

PDF only: (*New in v1.16.13*) Return a list of all XObjects referenced by a page.

Parameters `pno` (*int*) – page number, 0-based, `-inf < pno < pageCount`.

Return type `list`

Returns

a list of (non-image) XObjects. These objects typically represent pages *embedded* (not copied) from other PDFs. For example, `Page.showPDFpage()` will create this type of object. An item of this list has the following layout: **(xref, name, invoker, bbox)**, where

- `xref` (*int*) is the XObject's `xref`

² However, you **can** use `Document.getToC()` and `Page.getLinks()` (which are available for all document types) and copy this information over to the output PDF. See `demo/pdf-converter.py`.

- **name** (*str*) is the symbolic name to reference the XObject
- **invoker** (*int*) the *xref* of the invoking XObject or zero if the page directly invokes it
- **bbox** (*tuple*) the boundary box of the XObject's location on the page **in untransformed coordinates**. To get actual, non-rotated page coordinates, multiply with the page's transformation matrix *Page.transformationMatrix*.

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

PDF only: Return a list of all images (directly or indirectly) referenced by the page.

Parameters

- **pno** (*int*) – page number, 0-based, $-inf < pno < pageCount$.
- **full** (*bool*) – whether to also include the referencer's *xref* (which is zero if this is the page).

Return type

 list

Returns a list of images shown on this page. Each item looks like

(**xref**, **smask**, **width**, **height**, **bpc**, **colorspace**, **alt. colorspace**, **name**, **filter**, **referencer**)

Where

- **xref** (*int*) is the image object number
- **smask** (*int*) is the object number of its soft-mask image
- **width** and **height** (*ints*) are the image dimensions
- **bpc** (*int*) denotes the number of bits per component (normally 8)
- **colorspace** (*str*) a string naming the colorspace (like **DeviceRGB**)
- **alt. colorspace** (*str*) is any alternate colorspace depending on the value of **colorspace**
- **name** (*str*) is the symbolic name by which the image is referenced
- **filter** (*str*) is the decode filter of the image (*Adobe PDF References*, pp. 65).
- **referencer** (*int*) the *xref* of the referencer. Zero if directly referenced by the page. Only present if *full=True*.

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

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

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

PDF only: Return a list of all fonts (directly or indirectly) referenced by the page.

Parameters

- **pno** (*int*) – page number, 0-based, $-inf < pno < pageCount$.
- **full** (*bool*) – whether to also include the referencer's *xref*. If *True*, the returned items are one entry longer. Use this option if you need to know, whether the page directly references the font. In this case the last entry is 0. If the font is referenced by an /XObject of the page, you will find its *xref* here.

Return type list

Returns a list of fonts referenced by this page. Each entry looks like

(**xref**, **ext**, **type**, **basefont**, **name**, **encoding**, **referencer**),

where

- **xref** (*int*) is the font object number (may be zero if the PDF uses one of the builtin fonts directly)
- **ext** (*str*) font file extension (e.g. “ttf”, see *Font File Extensions*)
- **type** (*str*) is the font type (like “Type1” or “TrueType” etc.)
- **basefont** (*str*) is the base font name,
- **name** (*str*) is the symbolic name, by which the font is referenced
- **encoding** (*str*) the font’s character encoding if different from its built-in encoding (*Adobe PDF References*, p. 414):
- **referencer** (*int* optional) the *xref* of the referencer. Zero if directly referenced by the page, otherwise the xref of an XObject. Only present if *full=True*.

Example:

```
>>> pprint(doc.getPageFontList(0, full=False))
[(12, 'ttf', 'TrueType', 'FNUUTH+Calibri-Bold', 'R8', ''),
 (13, 'ttf', 'TrueType', 'DOKBTG+Calibri', 'R10', ''),
 (14, 'ttf', 'TrueType', 'NOHSJV+Calibri-Light', 'R12', ''),
 (15, 'ttf', 'TrueType', 'NZNDCL+CourierNewPSMT', 'R14', ''),
 (16, 'ttf', 'Type0', 'MNCSJY+SymbolMT', 'R17', 'Identity-H'),
 (17, 'cff', 'Type1', 'UAEUYH+Helvetica', 'R20', 'WinAnsiEncoding'),
 (18, 'ttf', 'Type0', 'ECPLRU+Calibri', 'R23', 'Identity-H'),
 (19, 'ttf', 'Type0', 'TONAYT+CourierNewPSMT', 'R27', 'Identity-H')]
```

Note: This list has no duplicate entries: the combination of *xref*, *name* and *referencer* is unique.

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

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

Parameters

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

Return type str

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

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

Parameters

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

- **fontsize** (*float*) – the desired default fontsize.

select (*s*)

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

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

Note:

- Page numbers in the sequence need not be unique nor be in any particular order. This makes the method a versatile utility to e.g. select only the even or the odd pages or meeting some other criteria and so forth.
- On a technical level, the method will always create a new *pagetree*.
- When dealing with only a few pages, methods *copyPage()*, *movePage()*, *deletePage()* are easier to use. In fact, they are also **much faster** – by at least one order of magnitude when the document has many pages.

setMetadata (*m*)

PDF only: Sets or updates the metadata of the document as specified in *m*, a Python dictionary.

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

(*Changed in v1.18.4*) Empty values or "none" are no longer written, but completely omitted.

getXmlMetadata ()

PDF only: Get the document XML metadata.

Return type str

Returns XML metadata of the document. Empty string if not present or not a PDF.

setXmlMetadata (*xml*)

PDF only: Sets or updates XML metadata of the document.

Parameters *xml* (*str*) – the new XML metadata. Should be XML syntax, however no checking is done by this method and any string is accepted.

setTOC (*toc*, *collapse=1*)**setTOC** (*toc*, *collapse=1*)

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

- Step 1 deletes all existing bookmarks.

- Step 2 creates a new TOC from the entries contained in *toc*.

Parameters

- **toc** (*sequence*) – A list or tuple with **all bookmark entries** that should form the new table of contents. Output variants of *getToC()* are acceptable. To completely remove the table of contents specify an empty sequence or None. Each item must be a list with the following format.
 - [lvl, title, page [, dest]] where
 - * **lvl** is the hierarchy level (int > 0) of the item, which **must be 1** for the first item and at most 1 larger than the previous one.
 - * **title** (str) is the title to be displayed. It is assumed to be UTF-8-encoded (relevant for multibyte code points only).
 - * **page** (int) is the target page number (**attention: 1-based**). Must be in valid range if positive. Set it 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 the page. Use a dictionary (like the one given as output by *getToC(False)*) if you want to store destinations that are either “named”, or reside outside this document (other files, internet resources, etc.).
- **collapse** (*int*) – (*new in version 1.16.9*) controls the hierarchy level beyond which outline entries should initially show up collapsed. The default 1 will hence only display level 1, higher levels must be expanded in the PDF viewer. To completely expand specify either a large integer, 0 or None.

Return type int

Returns the number of inserted, resp. deleted items.

outlineXref (*idx*)

(*New in v1.17.7*)

PDF only: Return the *xref* of the outline item. This is mainly used for internal purposes.

arg int *idx*: index of the item in list “*Document.getTOC*”.

Returns *xref*.

del_toc_item (*idx*)

(*New in v1.17.7*)

PDF only: Remove this TOC item. This is intended to be a high-speed method primarily meant for *disabling* items, which are pointing to deleted pages. Physically, the item still exists in the TOC tree, but will show an empty title and no longer point to a destination. So the overall TOC structure remains intact.

This also implies that you can reassign the item when required.

Parameters *idx* (*int*) – the index of the item in list *Document.getTOC*.

set_toc_item (*idx*, *dest_dict=None*, *kind=None*, *pno=None*, *uri=None*, *title=None*, *to=None*, *filename=None*, *zoom=0*)

(*New in v1.17.7*)

PDF only: Changes the TOC item identified by its index. It is possible to change the title text and the location this item is pointing to – or to remove the item as such.

Use this method if you need specific changes for selected entries only and want to avoid replacing the complete TOC. This is beneficial especially if you are dealing with large table of contents.

Parameters

- **idx** (*int*) – the index of the entry in the list created by `Document.getTOC()`.
- **dest_dict** (*dict*) – the new destination. A dictionary like the last entry of an item in `doc.getTOC(False)`. Using this as a template would also be the natural use of this parameter. When given, **all other parameters are ignored** – except title.
- **kind** (*int*) – the link kind, values like `fitz.LINK_GOTO`, etc. If equal to `fitz.LINK_NONE`, then all remaining parameter will be ignored, and the TOC item will be removed – same as `Document.del_toc_item()`. If `None`, then only the title is modified and the remaining parameters are ignored. All other values will lead to making a new destination dictionary using the subsequent arguments.
- **pno** (*int*) – the 1-based page number, i.e. a value $1 \leq pno \leq doc.pageCount$. Required for `LINK_GOTO`.
- **uri** (*str*) – the URL text. Required for `LINK_URI`.
- **title** (*str*) – the desired new title. None if no change.
- **to** (*point_like*) – (optional) points to a coordinate on the arget page. Relevant for `LINK_GOTO`. If omitted, a point near the page’s top is chosen.
- **filename** (*str*) – required for `LINK_GOTOR` and `LINK_LAUNCH`.
- **zoom** (*float*) – use this zoom factor when showing the target page.

`can_save_incrementally()`

(New in version 1.16.0)

Check whether the document can be saved incrementally. Use it to choose the right option without encountering exceptions.

scrub (*attached_files=True, clean_pages=True, embedded_files=True, hidden_text=True, javascript=True, metadata=True, redactions=True, redact_images=0, remove_links=True, reset_fields=True, reset_responses=True, xml_metadata=True*)

PDF only: (New in v1.16.14) Remove potentially sensitive data from the PDF. This function is inspired by the similar “Sanitize” function in Adobe Acrobat products. The process is configurable by a number of options, which are all *True* by default.

Parameters

- **attached_files** (*bool*) – Search for ‘FileAttachment’ annotations and remove the file content.
- **clean_pages** (*bool*) – Remove any comments from page painting sources. If this option is set to *False*, then this is also done for *hidden_text* and *redactions*.
- **embedded_files** (*bool*) – Remove embedded files.
- **hidden_text** (*bool*) – Remove OCR-ed text and invisible text.
- **javascript** (*bool*) – Remove JavaScript sources.
- **metadata** (*bool*) – Remove PDF standard metadata.
- **redactions** (*bool*) – Apply redaction annotations.
- **redact_images** (*int*) – how to handle images if applying redactions. One of 0 (ignore), 1 (blank out overlaps) or 2 (remove).
- **remove_links** (*bool*) – Remove all links.
- **reset_fields** (*bool*) – Reset all form fields to their defaults.

- **reset_responses** (*bool*) – Remove all responses from all annotations.
- **xml_metadata** (*bool*) – Remove XML metadata.

save (*outfile*, *garbage*=0, *clean*=False, *deflate*=False, *deflate_images*=False, *deflate_fonts*=False, *incremental*=False, *ascii*=False, *expand*=0, *linear*=False, *pretty*=False, *encryption*=PDF_ENCRYPT_NONE, *permissions*=-1, *owner_pw*=None, *user_pw*=None)
(Changed in v1.18.3)

PDF only: Saves the document in its **current state**.

Parameters

- **outfile** (*str*) – The file path to save to. Must be different from the original value if “incremental” is false or zero. When saving incrementally, “garbage” and “linear” **must be** false or zero and this parameter **must equal** the original filename (for convenience use *doc.name*).
- **garbage** (*int*) – Do garbage collection. Positive values exclude “incremental”.
 - 0 = none
 - 1 = remove unused (unreferenced) objects.
 - 2 = in addition to 1, compact the *xref* table.
 - 3 = in addition to 2, merge duplicate objects.
 - 4 = in addition to 3, check *stream* objects for duplication. This may be slow because such data are typically large.
- **clean** (*bool*) – Clean and sanitize content streams¹. Corresponds to “mutool clean -sc”.
- **deflate** (*bool*) – Deflate (compress) uncompressed streams.
- **deflate_images** (*bool*) – (*new in v1.18.3*) Deflate (compress) uncompressed image streams⁴.
- **deflate_fonts** (*bool*) – (*new in v1.18.3*) Deflate (compress) uncompressed fontfile streams⁴.
- **incremental** (*bool*) – Only save changed objects. Excludes “garbage” and “linear”. Cannot be used for files that are decrypted or repaired and also in some other cases. To be sure, check *Document.can_save_incrementally()*. If this is false, saving to a new file is required.
- **ascii** (*bool*) – convert binary data to ASCII.
- **expand** (*int*) – Decompress objects. Generates versions that can be better read by some other programs and will lead to larger files.
 - 0 = none
 - 1 = images
 - 2 = fonts
 - 255 = all

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 References*), which gets interpreted when a page is loaded.

⁴ These parameters cause separate handling of stream categories: use it together with `expand` to restrict decompression to streams other than images / fontfiles.

- **linear** (*bool*) – Save a linearised version of the document. This option creates a file format for improved performance when read via internet connections. Excludes “incremental”.
- **pretty** (*bool*) – Prettify the document source for better readability. PDF objects will be reformatted to look like the default output of `Document.xrefObject()`.
- **permissions** (*int*) – (new in version 1.16.0) Set the desired permission levels. See *Document Permissions* for possible values. Default is granting all.
- **encryption** (*int*) – (new in version 1.16.0) set the desired encryption method. See *PDF encryption method codes* for possible values.
- **owner_pw** (*str*) – (new in version 1.16.0) set the document’s owner password. (Changed in v1.18.3) If not provided, the user password is taken if provided.
- **user_pw** (*str*) – (new in version 1.16.0) set the document’s user password.

saveIncr()

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

write (*garbage=0, clean=False, deflate=False, deflate_images=False, deflate_fonts=False, ascii=False, expand=0, pretty=False, encryption=PDF_ENCRYPT_NONE, permissions=-1, owner_pw=None, user_pw=None*)
(Changed in v1.18.3)

PDF only: Writes the **current content of the document** to a bytes object instead of to a file. Obviously, you should be wary about memory requirements. The meanings of the parameters exactly equal those in `save()`. Chapter *Collection of Recipes* contains an example for using this method as a pre-processor to `pdfw`.

(Changed in version 1.16.0) for extended encryption support.

Return type bytes

Returns a bytes object containing the complete document.

searchPageFor (*pno, text, quads=False*)

Search for “text” on page number “pno”. Works exactly like the corresponding `Page.searchFor()`. Any integer $-\text{inf} < \text{pno} < \text{pageCount}$ is acceptable.

insertPDF (*docsrc, from_page=-1, to_page=-1, start_at=-1, rotate=-1, links=True, annots=True, show_progress=0, final=1*)

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`. Value -1 indicates default values. All pages thus copied will be rotated as specified. Links and annotations can be excluded in the target, see below. All page numbers are 0-based.

Parameters

- **docsrc** (*Document*) – An opened PDF *Document* which must not be the current document. 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. Defaults to last page.
- **start_at** (*int*) – First copied page, will become page number `start_at` in the target. Default -1 appends the page range to the end. If zero, the page range will be inserted before current first page.
- **rotate** (*int*) – All copied pages will be rotated by the provided value (degrees, integer multiple of 90).

- **links** (*bool*) – Choose whether (internal and external) links should be included in the copy. Default is *True*. Internal links to outside the copied page range are **always excluded**.
- **annots** (*bool*) – (*new in version 1.16.1*) choose whether annotations should be included in the copy.
- **show_progress** (*int*) – (*new in version 1.17.7*) specify an interval size greater zero to see progress messages on `sys.stdout`. After each interval, a message like `Inserted 30 of 47 pages.` will be printed.
- **final** (*int*) – (*new in v1.18.0*) controls whether the list of already copied objects should be **dropped** after this method, default *True*. Set it to 0 except for the last one of multiple insertions from the same source PDF. This saves target file size and speeds up execution considerably.

Note:

1. If `from_page > to_page`, pages will be **copied in reverse order**. If `0 <= from_page == to_page`, then one page will be copied.
 2. `docsrc` TOC entries **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.
-

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

PDF only: Insert an empty page.

Parameters

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

Return type *Page*

Returns the created page object.

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

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

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

Changed in version 1.14.12 This is now a positional parameter

For the other parameters, please consult the aforementioned methods.

Return type *int*

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

deletePage (*pno=-1*)

PDF only: Delete a page given by its 0-based number in `-inf < pno < pageCount - 1`.

Changed in version 1.14.17

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

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

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

Parameters

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

Note: (*Changed in v1.14.17, optimized in v1.17.7*) In an effort to maintain a valid PDF structure, this method and `deletePage()` will also invalidate items in the table of contents which happen to point to deleted pages. “Invalidation” here means, that the bookmark will point to nowhere and the title will show the string “<>”. So the overall TOC structure is left intact.

Similarly, it will remove any **links on remaining pages** that point to a deleted page. This action may have an extended response time for documents with many pages.

Example: Delete the page range 500 to 520 from a large PDF, using different methods.

Method 1 - `deletePageRange`:

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

Method 2 - `select`, this is more than 10 times **slower**:

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

copyPage (*pno, to=-1*)

PDF only: Copy a page reference within the document.

Parameters

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

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

fullcopyPage (*pno, to=-1*)

(*New in version 1.14.17*)

PDF only: Make a full copy (duplicate) of a page.

Parameters

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

Note:

- In contrast to `copyPage()`, this method creates a new page object (with a new *xref*), which can be changed independently from the original.
 - Any Popup and “IRT” (“in response to”) annotations are **not copied** to avoid potentially incorrect situations.
-

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

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

Parameters

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

need_appearances (*value=None*)

(New in v1.17.4)

PDF only: Get or set the `/NeedAppearances` property of Form PDFs. Quote: “(Optional) A flag specifying whether to construct appearance streams and appearance dictionaries for all widget annotations in the document ... Default value: false.” This may help controlling the behavior of some readers / viewers.

Parameters **value** (*bool*) – set the property to this value. If omitted or *None*, inquire the current value.

Return type *bool*

Returns

- None: not a Form PDF or property not defined.
- True / False: the value of the property (either just set or existing for inquiries).

Once set, the property cannot be removed again (which is no problem).

getSigFlags ()

PDF only: Return whether the document contains signature fields. This is an optional PDF property: if not present (return value -1), no conclusions can be drawn – the PDF creator may just not have bothered to use it.

Return type *int*

Returns

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

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

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

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

Parameters

- **name** (*str*) – entry identifier, must not already exist.
- **buffer** (*bytes, bytearray, BytesIO*) – file contents.
(*Changed in version 1.14.13*) *io.BytesIO* is now also supported.
- **filename** (*str*) – optional filename. Documentation only, will be set to *name* if *None*.
- **ufilename** (*str*) – optional unicode filename. Documentation only, will be set to *filename* if *None*.
- **desc** (*str*) – optional description. Documentation only, will be set to *name* if *None*.

embeddedFileCount ()

PDF only: Return the number of embedded files.

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

embeddedFileGet (*item*)

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

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

Return type bytes

embeddedFileDel (*item*)

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

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

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

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

embeddedFileInfo (*item*)

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

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

Return type dict

Returns

a dictionary with the following keys:

- *name* – (*str*) name under which this entry is stored
- *filename* – (*str*) filename

- *ufilename* – (unicode) filename
- *desc* – (str) description
- *size* – (int) original file size
- *length* – (int) compressed file length

embeddedFileNames ()

(New in version 1.14.16)

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

Return type list

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

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

Parameters

- **item** (*int/str*) – index or name of entry. An integer must be in *range(0, embeddedFileCount())*.
- **buffer** (*bytes, bytearray, BytesIO*) – the new file content.
(Changed in version 1.14.13) *io.BytesIO* is now also supported.
- **filename** (*str*) – the new filename.
- **ufilename** (*str*) – the new unicode filename.
- **desc** (*str*) – the new description.

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

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

Parameters

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

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

close ()

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

xrefObject (*xref*, *compressed=False*, *ascii=False*)

(New in version 1.16.8)

PDF only: Return the definition of a PDF object. For details please refer to *Document.xrefObject()*.

PDFCatalog ()

(New in version 1.16.8)

PDF only: Return the *xref* of the PDF catalog (or root) object. For details please refer to `Document._getPDFroot()`.

PDFTrailer (*compressed=False*)

(New in version 1.16.8)

PDF only: Return the trailer of the PDF (UTF-8), which is usually located at the PDF file's end. For details please refer to `Document._getTrailerString()`.

metadataXML ()

(New in version 1.16.8)

PDF only: Return the *xref* of the document's XML metadata. For details please refer to `Document._getXmlMetadataXref()`.

xrefStream (*xref*)

(New in version 1.16.8)

PDF only: Return the **decompressed** contents of the *xref* stream object.

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

Return type bytes

Returns the (decompressed) stream of the object.

xrefStreamRaw (*xref*)

(New in version 1.16.8)

PDF only: Return the **unmodified** (esp. **not decompressed**) contents of the *xref* stream object. Otherwise equal to `Document.xrefStream()`.

Return type bytes

Returns the (original) stream of the object.

updateObject (*xref, obj_str, page=None*)

(New in version 1.16.8)

PDF only: Replace object definition of *xref* with the provided string. The *xref* may also be new, in which case this instruction completes the object definition. If a page object is also given, its links and annotations will be reloaded afterwards.

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.

updateStream (*xref, data, new=False*)

(New in version 1.16.8)

Replace the stream of an object identified by *xref*. If the object has no stream, an exception is raised unless *new=True* is used. The function automatically performs a compress operation (“deflate”) where beneficial.

Parameters

- **xref** (*int*) – *xref* number.
- **stream** (*bytes/bytearray/BytesIO*) – the new content of the stream.
(*Changed in version 1.14.13:*) *io.BytesIO* objects are now also supported.
- **new** (*bool*) – whether to force accepting the stream, and thus **turning it into a stream object**.

This method is intended to manipulate streams containing PDF operator syntax (see pp. 985 of the *Adobe PDF References*) as it is the case for e.g. page content streams.

If you update a contents stream, you should use save parameter *clean=True*. This ensures consistency between PDF operator source and the object structure.

Example: Let us assume that you no longer want a certain image appear on a page. This can be achieved by deleting the respective reference in its contents source(s) – and indeed: the image will be gone after reloading the page. But the page's *resources* object would still show the image as being referenced by the page. This save option will clean up any such mismatches.

outline

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

Type *Outline*

isClosed

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

Type bool

isPDF

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

Type bool

isFormPDF

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

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

Type bool,int

isReflowable

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

Type bool

isRepaired

(*New in v1.18.2*)

True if PDF has been repaired during open (because of major structure issues). Always *False* for non-PDF documents. If true, more details have been stored in *TOOLS.mupdf_warnings()*, and *Document.can_save_incrementally()* will return *False*.

Type bool

needsPass

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

Type bool

isEncrypted

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

Type bool

permissions

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

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

Type int

metadata

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

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

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

Type dict

name

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

Type str

pageCount

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

Type int

chapterCount

(New in version 1.17.0) Contains the number of chapters in the document. Always at least 1. Relevant only for document types with chapter support (EPUB currently). Other documents will return 1.

Type int

lastLocation

(New in version 1.17.0) Contains (chapter, pno) of the document's last page. Relevant only for document types with chapter support (EPUB currently). Other documents will return (0, len(doc) - 1) and (0, -1) if it has no pages.

Type int

FormFonts

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

Type list

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

6.4.1 setMetadata() Example

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

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

6.4.2 setToC() Demonstration

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()
```

(continues on next page)

(continued from previous page)

```

>>> 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]

```

6.4.3 insertPDF() Examples

(1) Concatenate two documents including their TOCs:

```

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

```

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

```

6.4.4 Other Examples

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

```

for i in range(len(doc)):
    imglist = doc.getPageImageList(i)
    for img in imglist:
        xref = img[0]                                     # xref number
        pix = fitz.Pixmap(doc, xref)                     # make pixmap from image

```

(continues on next page)

(continued from previous page)

```

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)
```

6.5 Font

(New in v1.16.18) This class represents a font as defined in MuPDF (*fz_font_s* structure). It is required for the new class *TextWriter* and the new *Page.writeText()*. Currently, it has no connection to how fonts are used in methods *insertText* or *insertTextbox*“, respectively.

A Font object also contains useful general information, like the font bbox, the number of defined glyphs, glyph names or the bbox of a single glyph.

Method / Attribute	Short Description
<i>glyph_advance()</i>	Width of a character
<i>glyph_bbox()</i>	Glyph rectangle
<i>glyph_name_to_unicode()</i>	Get unicode from glyph name
<i>has_glyph()</i>	Return glyph id of unicode
<i>text_length()</i>	Compute text length under a fontsize
<i>unicode_to_glyph_name()</i>	Get glyph name of a unicode
<i>valid_codepoints()</i>	Array of supported unicodes
<i>ascender</i>	Font ascender
<i>descender</i>	Font descender
<i>bbox</i>	Font rectangle
<i>buffer</i>	Copy of the font’s binary image
<i>flags</i>	Collection of font properties
<i>glyph_count</i>	Number of supported glyphs
<i>name</i>	Name of font
<i>isWritable</i>	Font usable with <i>TextWriter</i>

Class API**class Font**

```

__init__(self, fontname=None, fontfile=None,
fontbuffer=None, script=0, language=None, ordering=-1, is_bold=0,
is_italic=0, is_serif=0)

```

Font constructor. The large number of parameters are used to locate font, which most closely resembles the requirements. Not all parameters are ever required – see the below pseudo code explaining the logic how the parameters are evaluated.

Parameters

- **fontname** (*str*) – one of the *PDF Base 14 Fonts* or CJK fontnames. Also possible are a select few other names like (watch the correct spelling): “Arial”, “Times”, “Times Roman”.

(Changed in v1.17.5)

If you have installed `pymupdf-fonts`, there are also new “reserved” fontnames available, which are listed in `fitz_fonts` and in the table further down.

- **filename** (*str*) – the filename of a fontfile somewhere on your system¹.
- **fontbuffer** (*bytes, bytearray, io.BytesIO*) – a fontfile loaded in memory¹.
- **script** (*in*) – the number of a UCDN script. Currently supported in PyMuPDF are numbers 24, and 32 through 35.
- **language** (*str*) – one of the values “zh-Hant” (traditional Chinese), “zh-Hans” (simplified Chinese), “ja” (Japanese) and “ko” (Korean). Otherwise, all ISO 639 codes from the subsets 1, 2, 3 and 5 are also possible, but are currently documentary only.
- **ordering** (*int*) – an alternative selector for one of the CJK fonts.
- **is_bold** (*bool*) – look for a bold font.
- **is_italic** (*bool*) – look for an italic font.
- **is_serif** (*bool*) – look for a serified font.

Returns

a MuPDF font if successful. This is the overall sequence of checks to determine an appropriate font:

Argument	Action
fontfile?	Create font from file, exception if failure.
font-buffer?	Create font from buffer, exception if failure.
ordering>=0	Create universal font, always succeeds.
font-name?	Create a Base-14 font, universal font, or font provided by <code>pymupdf-fonts</code> . See table below.

Note: With the usual reserved names “helv”, “tiro”, etc., you will create fonts with the expected names “Helvetica”, “Times-Roman” and so on. **However**, and in contrast to `Page.insertFont()` and friends,

- a font file will **always** be embedded in your PDF,
- Greek and Cyrillic characters are supported without needing the *encoding* parameter.

Using *ordering* `>= 0`, or fontnames “`CJK`”, “`china-t`”, “`china-s`”, “`japan`” or “`korea`” will **always create the same “universal” font “Droid Sans Fallback Regular”**. This font supports **all CJK and all Latin characters**, including Greek and Cyrillic.

¹ MuPDF does not support all fontfiles with this feature and will raise exceptions like “`mupdf: FT_New_Memory_Face((null)): unknown file format`”, if it encounters issues.

Actually, you would rarely ever need another font than “**Droid Sans Fallback Regular**”. **Except** that this font file is relatively large and adds about 1.65 MB (compressed) to your PDF file size. If you do not need CJK support, stick with specifying “`helv`”, “`tiro`” etc., and you will get away with about 35 KB compressed.

If you **know** you have a mixture of CJK and Latin text, consider just using `Font("cjk")` because this supports everything and also significantly (by a factor of two to three) speeds up execution: MuPDF will always find any character in this single font and need not check fallbacks.

But if you do specify a Base-14 fontname, you will still be able to also write CJK characters: MuPDF detects this situation and silently falls back to the universal font (which will then of course also be embedded in your PDF).

(*New in v1.17.5*) Optionally, some new “reserved” fontname codes become available if you install `pymupdf-fonts`. “**Fira Mono**” is a nice mono-spaced sans font set and **FiraGO** is another non-serifed “universal” font, set which supports all Latin (including Cyrillic and Greek) plus Thai, Arabian, Hebrew and Devanagari – but none of the CJK languages. The size of a FiraGO font is only a quarter of the “Droid Sans Fallback” size (compressed 400 KB vs. 1.65 MB) – **and** it provides the weight bold, italic, bold-italic – which the universal font doesn’t.

“**Space Mono**” is another nice and small mono-spaced font from Google Fonts, which supports Latin Extended characters and comes with all 4 important weights.

The following table maps a fontname code to the corresponding font:

Code	Fontname	New in	Comment
<code>figo</code>	FiraGO Regular	v1.0.0	narrower than Helvetica
<code>figbo</code>	FiraGO Bold	v1.0.0	
<code>figit</code>	FiraGO Italic	v1.0.0	
<code>figbi</code>	FiraGO Bold Italic	v1.0.0	
<code>fimo</code>	Fira Mono Regular	v1.0.0	
<code>fimbo</code>	Fira Mono Bold	v1.0.0	
<code>spacemo</code>	Space Mono Regular	v1.0.1	
<code>spacembo</code>	Space Mono Bold	v1.0.1	
<code>spacemit</code>	Space Mono Italic	v1.0.1	
<code>spacembi</code>	Space Mono Bold-Italic	v1.0.1	
<code>math</code>	Noto Sans Math Regular	v1.0.2	math symbols
<code>music</code>	Noto Music Regular	v1.0.2	musical symbols
<code>symbol1</code>	Noto Sans Symbols Regular	v1.0.2	replacement for “ <code>symb</code> ”
<code>symbol2</code>	Noto Sans Symbols2 Regular	v1.0.2	extended symbol set
<code>notos</code>	Noto Sans Regular	v1.0.3	alternative to Helvetica
<code>notosit</code>	Noto Sans Italic	v1.0.3	
<code>notosbo</code>	Noto Sans Bold	v1.0.3	
<code>notosbi</code>	Noto Sans BoldItalic	v1.0.3	

has_glyph (*chr*, *language=None*, *script=0*, *fallback=False*)

Check whether the unicode *chr* exists in the font or some fallback font. May be used to check whether any “TOFU” symbols will appear on output.

Parameters

- **chr** (*int*) – the unicode of the character (i.e. `ord()`).
- **language** (*str*) – the language – currently unused.
- **script** (*int*) – the UCDN script number.

- **fallback** (*bool*) – (new in v1.17.5) perform an extended search in fallback fonts or restrict to current font (default).

Returns (changed in 1.17.7) the glyph number. Zero indicates no glyph found.

valid_codepoints ()

(New in v1.17.5)

Return an array of unicones supported by this font.

Returns

an *array.array*² of length at most *Font.glyph_count*. I.e. *chr()* of every item in this array has a glyph in the font without using fallbacks. This is an example display of the supported glyphs:

```
>>> import fitz
>>> font = fitz.Font("math")
>>> vuc = font.valid_codepoints()
>>> for i in vuc:
    print("%04X %s (%s)" % (i, chr(i), font.unicode_to_glyph_
↳name(i)))
0000
000D (CR)
0020 (space)
0021 ! (exclam)
0022 " (quotedbl)
0023 # (numbersign)
0024 $ (dollar)
0025 % (percent)
...
00AC ¬ (logicalnot)
00B1 ± (plusminus)
...
21D0 (arrowdblleft)
21D1 (arrowdblup)
21D2 (arrowdblright)
21D3 (arrowdbldown)
21D4 (arrowdblboth)
...
221E ∞ (infinity)
...
```

Note: This method only returns meaningful data for fonts having a CMAP (character map, charmap, the *ToUnicode* PDF key). Otherwise, this array will have length 1 and contain zero only.

glyph_advance (*chr*, *language=None*, *script=0*, *wmode=0*)

Calculate the “width” of the character’s glyph (visual representation).

Parameters

- **chr** (*int*) – the unicode number of the character. Use *ord()*, not the character itself. Again, this should normally work even if a character is not supported by that font, because fallback fonts will be checked where necessary.
- **wmode** (*int*) – write mode, 0 = horizontal, 1 = vertical.

The other parameters are not in use currently.

² The built-in module *array* has been chosen for its speed and its compact representation of values.

Returns a float representing the glyph's width relative to **fontsize 1**.

glyph_name_to_unicode (*name*)

Return the unicode value for a given glyph name. Use it in conjunction with `chr()` if you want to output e.g. a certain symbol.

Parameters **name** (*str*) – The name of the glyph.

Returns

The unicode integer, or 65533 = 0xFFFFD if the name is unknown. Examples: `font.glyph_name_to_unicode("Sigma") = 931`, `font.glyph_name_to_unicode("sigma") = 963`. Refer to the [Adobe Glyph List](#) publication for a list of glyph names and their unicode numbers. Example:

```
>>> font = fitz.Font("helv")
>>> font.has_glyph(font.glyph_name_to_unicode("infinity"))
True
```

glyph_bbox (*chr*, *language=None*, *script=0*)

The glyph rectangle relative to **fontsize 1**.

Parameters **chr** (*int*) – *ord()* of the character.

Returns a *Rect*.

unicode_to_glyph_name (*ch*)

Show the name of the character's glyph.

Parameters **ch** (*int*) – the unicode number of the character. Use *ord()*, not the character itself.

Returns

a string representing the glyph's name. E.g. `font.glyph_name(ord("#")) = "numbersign"`. For an invalid code "notfound" is returned.

Note: (*Changed in v1.18.0*) This method and `Font.glyph_name_to_unicode()` no longer depend on a font and instead retrieve information from the **Adobe Glyph List**. Also available as `fitz.unicode_to_glyph_name()` and resp. `fitz.glyph_name_to_unicode()`.

text_length (*text*, *fontsize=11*)

Calculate the length of a unicode string.

Parameters

- **text** (*str*) – a text string – UTF-8 encoded. For Python 2, you must use unicode here.
- **fontsize** (*float*) – the fontsize.

Return type float

Returns the length of the string when stored in the PDF. Internally `glyph_advance()` is used on a by-character level. If the font does not have a character, it will automatically be looked up in a fallback font.

buffer

(*New in v1.17.6*)

Return type bytes

Copy of the binary font file content.

flags

Return type dict

A dictionary with various font properties, each represented as bools. Example for Helvetica:

```
>>> pprint(font.flags)
{'bold': 0,
 'fake-bold': 0,
 'fake-italic': 0,
 'invalid-bbox': 0,
 'italic': 0,
 'mono': 0,
 'opentype': 0,
 'serif': 1,
 'stretch': 0,
 'substitute': 0}
```

name

Return type str

Name of the font. May be "" or "(null)".

bbox

Return type *Rect*

The font bbox. This is the maximum of its glyph bboxes.

glyph_count

Return type int

The number of glyphs defined in the font.

ascender

(New in v1.18.0)

Return type float

The ascender value of the font, see [here](#) for details.

descender

(New in v1.18.0)

Return type float

The descender value of the font, see [here](#) for details.

isWritable

(New in v1.18.0)

Return type bool

Indicates whether this font can be used with *TextWriter*.

6.6 Identity

Identity is a *Matrix* that performs no action – to be used whenever the syntax requires a matrix, but no actual transformation should take place. It has the form *fitz.Matrix(1, 0, 0, 1, 0, 0)*.

Identity is a constant, an “immutable” object. So, all of its matrix properties are read-only and its methods are disabled.

If you need a **mutable** identity matrix as a starting point, use one of the following statements:

```
>>> m = fitz.Matrix(1, 0, 0, 1, 0, 0) # specify the values
>>> m = fitz.Matrix(1, 1)           # use scaling by factor 1
>>> m = fitz.Matrix(0)             # use rotation by zero degrees
>>> m = fitz.Matrix(fitz.Identity) # make a copy of Identity
```

6.7 IRect

IRect is a rectangular bounding box similar to *Rect*, except that all corner coordinates are integers. IRect is used to specify an area of pixels, e.g. to receive image data during rendering. Otherwise, many similarities exist, e.g. considerations concerning emptiness and finiteness of rectangles also apply to this class.

Attribute / Method	Short Description
<i>IRect.contains()</i>	checks containment of another object
<i>IRect.getArea()</i>	calculate rectangle area
<i>IRect.getRect()</i>	return a <i>Rect</i> with same coordinates
<i>IRect.getRectArea()</i>	calculate rectangle area
<i>IRect.intersect()</i>	common part with another rectangle
<i>IRect.intersects()</i>	checks for non-empty intersection
<i>IRect.morph()</i>	transform with a point and a matrix
<i>IRect.norm()</i>	the Euclidean norm
<i>IRect.normalize()</i>	makes a rectangle finite
<i>IRect.bottom_left</i>	bottom left point, synonym <i>bl</i>
<i>IRect.bottom_right</i>	bottom right point, synonym <i>br</i>
<i>IRect.height</i>	height of the rectangle
<i>IRect.isEmpty</i>	whether rectangle is empty
<i>IRect.isInfinite</i>	whether rectangle is infinite
<i>IRect.rect</i>	equals result of method <i>getRect()</i>
<i>IRect.top_left</i>	top left point, synonym <i>tl</i>
<i>IRect.top_right</i>	top_right point, synonym <i>tr</i>
<i>IRect.quad</i>	<i>Quad</i> made from rectangle corners
<i>IRect.width</i>	width of the rectangle
<i>IRect.x0</i>	X-coordinate of the top left corner
<i>IRect.x1</i>	X-coordinate of the bottom right corner
<i>IRect.y0</i>	Y-coordinate of the top left corner
<i>IRect.y1</i>	Y-coordinate of the bottom right corner

Class API

class IRect

`__init__(self)`

`__init__(self, x0, y0, x1, y1)`

`__init__(self, irect)`

`__init__(self, sequence)`

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

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

If sequence is specified, it must be a Python sequence type of 4 numbers (see *Using Python Sequences as Arguments in PyMuPDF*). Non-integer numbers will be truncated, non-numeric entries will raise an exception.

The other parameters mean integer coordinates.

getRect ()

A convenience function returning a *Rect* with the same coordinates. Also available as attribute *rect*.

Return type *Rect*

getRectArea ([*unit*])

getArea ([*unit*])

Calculates the area of the rectangle and, with no parameter, equals *abs(IRect)*. Like an empty rectangle, the area of an infinite rectangle is also zero.

Parameters **unit** (*str*) – Specify required unit: respective squares of “px” (pixels, default), “in” (inches), “cm” (centimeters), or “mm” (millimeters).

Return type float

intersect (*ir*)

The intersection (common rectangular area) of the current rectangle and *ir* is calculated and replaces the current rectangle. If either rectangle is empty, the result is also empty. If either rectangle is infinite, the other one is taken as the result – and hence also infinite if both rectangles were infinite.

Parameters **ir** (*rect_like*) – Second rectangle.

contains (*x*)

Checks whether *x* is contained in the rectangle. It may be *rect_like*, *point_like* or a number. If *x* is an empty rectangle, this is always true. Conversely, if the rectangle is empty this is always *False*, if *x* is not an empty rectangle and not a number. If *x* is a number, it will be checked to be one of the four components. *x in irect* and *irect.contains(x)* are equivalent.

Parameters **x** (*IRect* or *Rect* or *Point* or int) – the object to check.

Return type bool

intersects (*r*)

Checks whether the rectangle and the *rect_like* “*r*” contain a common non-empty *IRect*. This will always be *False* if either is infinite or empty.

Parameters **r** (*rect_like*) – the rectangle to check.

Return type bool

morph (*fixpoint*, *matrix*)

(New in version 1.17.0)

Return a new quad after applying a matrix to it using a fixed point.

Parameters

- **fixpoint** (*point_like*) – the fixed point.
- **matrix** (*matrix_like*) – the matrix.

Returns a new *Quad*. This a wrapper of the same-named quad method.

norm ()

(New in version 1.16.0)

Return the Euclidean norm of the rectangle treated as a vector of four numbers.

normalize()

Make the rectangle finite. This is done by shuffling rectangle corners. After this, the bottom right corner will indeed be south-eastern to the top left one. See [Rect](#) for a more details.

top_left

tl

Equals *Point(x0, y0)*.

Type *Point*

top_right

tr

Equals *Point(x1, y0)*.

Type *Point*

bottom_left

bl

Equals *Point(x0, y1)*.

Type *Point*

bottom_right

br

Equals *Point(x1, y1)*.

Type *Point*

quad

The quadrilateral *Quad(irect.tl, irect.tr, irect.bl, irect.br)*.

Type *Quad*

width

Contains the width of the bounding box. Equals *abs(x1 - x0)*.

Type *int*

height

Contains the height of the bounding box. Equals *abs(y1 - y0)*.

Type *int*

x0

X-coordinate of the left corners.

Type *int*

y0

Y-coordinate of the top corners.

Type *int*

x1

X-coordinate of the right corners.

Type *int*

y1

Y-coordinate of the bottom corners.

Type *int*

isInfinite*True* if rectangle is infinite, *False* otherwise.**Type** bool**isEmpty***True* if rectangle is empty, *False* otherwise.**Type** bool**Note:**

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
- Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.

6.8 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
<code>Link.setBorder()</code>	modify border properties
<code>Link.setColors()</code>	modify color properties
<code>Link.border</code>	border characteristics
<code>Link.colors</code>	border line color
<code>Link.dest</code>	points to link destination details
<code>Link.isExternal</code>	external link destination?
<code>Link.next</code>	points to next link
<code>Link.rect</code>	clickable area in untransformed coordinates.
<code>Link.uri</code>	link destination
<code>Link.xref</code>	<i>xref</i> number of the entry

Class API**class Link****setBorder** (*border=None, width=0, style=None, dashes=None*)

PDF only: Change border width and dashing properties.

(*Changed in version 1.16.9*) Allow specification without using a dictionary. The direct parameters are used if *border* is not a dictionary.

Parameters

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

- **width** (*float*) – see above.
- **style** (*str*) – see above.
- **dashes** (*sequence*) – see above.

setColors (*colors=None, stroke=None, fill=None*)

Changes the “stroke” and “fill” colors.

(*Changed in version 1.16.9*) Allow colors to be directly set. These parameters are used if *colors* is not a dictionary.

Parameters

- **colors** (*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.
- **stroke** (*sequence*) – see above.
- **fill** (*sequence*) – see above.

colors

Meaningful for PDF only: A dictionary of two lists of floats in range $0 \leq \text{float} \leq 1$ specifying the *stroke* and the interior (*fill*) colors. If not a PDF, *None* is returned. The stroke color is used for borders and everything that is actively painted or written (“stroked”). The lengths of these lists implicitly determine the colorspaces used: 1 = GRAY, 3 = RGB, 4 = CMYK. So $[1.0, 0.0, 0.0]$ stands for RGB color red. Both lists can be $[]$ if no color is specified. The value of each float f is mapped to the integer value i in range 0 to 255 via the computation $f = i / 255$.

Return type dict

border

Meaningful for PDF only: A dictionary containing border characteristics. It will be *None* for non-PDFs and an empty dictionary if no border information exists. The following keys can occur:

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

Return type dict

rect

The area that can be clicked in untransformed coordinates.

Type *Rect*

isExternal

A bool specifying whether the link target is outside of the current document.

Type bool

uri

A string specifying the link target. The meaning of this property should be evaluated in conjunction with property *isExternal*. The value may be *None*, in which case *isExternal* == *False*. If *uri* starts with *file://*,

mailto:, or an internet resource name, *isExternal* is *True*. In all other cases *isExternal* == *False* and *uri* points to an internal location. In case of PDF documents, this should either be *#nnnn* to indicate a 1-based (!) page number *nnnn*, or a named location. The format varies for other document types, e.g. *uri* = *'./FixedDoc.fdoc#PG_2_LNK_1'* for page number 2 (1-based) in an XPS document.

Type str

xref

An integer specifying the PDF *xref*. Zero if not a PDF.

Type int

next

The next link or *None*.

Type *Link*

dest

The link destination details object.

Type *linkDest*

6.9 linkDest

Class representing the *dest* property of an outline entry or a link. Describes the destination to which such entries point.

Attribute	Short Description
<i>linkDest.dest</i>	destination
<i>linkDest.fileSpec</i>	file specification (path, filename)
<i>linkDest.flags</i>	descriptive flags
<i>linkDest.isMap</i>	is this a MAP?
<i>linkDest.isUri</i>	is this a URI?
<i>linkDest.kind</i>	kind of destination
<i>linkDest.lt</i>	top left coordinates
<i>linkDest.named</i>	name if named destination
<i>linkDest.newWindow</i>	name of new window
<i>linkDest.page</i>	page number
<i>linkDest.rb</i>	bottom right coordinates
<i>linkDest.uri</i>	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 References](#)). 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

6.10 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 References](#)). 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 References*.

Method / Attribute	Description
<code>Matrix.preRotate()</code>	perform a rotation
<code>Matrix.preScale()</code>	perform a scaling
<code>Matrix.preShear()</code>	perform a shearing (skewing)
<code>Matrix.preTranslate()</code>	perform a translation (shifting)
<code>Matrix.concat()</code>	perform a matrix multiplication
<code>Matrix.invert()</code>	calculate the inverted matrix
<code>Matrix.norm()</code>	the Euclidean norm
<code>Matrix.a</code>	zoom factor X direction
<code>Matrix.b</code>	shearing effect Y direction
<code>Matrix.c</code>	shearing effect X direction
<code>Matrix.d</code>	zoom factor Y direction
<code>Matrix.e</code>	horizontal shift
<code>Matrix.f</code>	vertical shift
<code>Matrix.isRectilinear</code>	true if rect corners will remain rect corners

Class API

class Matrix

```

__init__(self)
__init__(self, zoom-x, zoom-y)
__init__(self, shear-x, shear-y, 1)
__init__(self, a, b, c, d, e, f)
__init__(self, matrix)
__init__(self, degree)

```

`__init__` (*self, sequence*)

Overloaded constructors.

Without parameters, the zero matrix *Matrix(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)* will be created.

*zoom-** and *shear-** specify zoom or shear values (float) and create a zoom or shear matrix, respectively.

For “matrix” a **new copy** of another matrix will be made.

Float value “degree” specifies the creation of a rotation matrix which rotates anit-clockwise.

A “sequence” must be any Python sequence object with exactly 6 float entries (see *Using Python Sequences as Arguments in PyMuPDF*).

fitz.Matrix(1, 1), *fitz.Matrix(0.0)* and *fitz.Matrix(fitz.Identity)* create modifyable versions of the *Identity* matrix, which looks like *[1, 0, 0, 1, 0, 0]*.

`norm` ()

(*New in version 1.16.0*)

Return the Euclidean norm of the matrix as a vector.

`preRotate` (*deg*)

Modify the matrix to perform a counter-clockwise rotation for positive *deg* degrees, else clockwise. The matrix elements of an identity matrix will change in the following way:

[1, 0, 0, 1, 0, 0] -> *[cos(deg), sin(deg), -sin(deg), cos(deg), 0, 0]*.

Parameters *deg* (*float*) – The rotation angle in degrees (use conventional notation based on Pi = 180 degrees).

`preScale` (*sx, sy*)

Modify the matrix to scale by the zoom factors *sx* and *sy*. Has effects on attributes *a* thru *d* only: *[a, b, c, d, e, f]* -> *[a*sx, b*sx, c*sy, d*sy, e, f]*.

Parameters

- ***sx*** (*float*) – Zoom factor in X direction. For the effect see description of attribute *a*.
- ***sy*** (*float*) – Zoom factor in Y direction. For the effect see description of attribute *d*.

`preShear` (*sx, sy*)

Modify the matrix to perform a shearing, i.e. transformation of rectangles into parallelograms (rhomboids). Has effects on attributes *a* thru *d* only: *[a, b, c, d, e, f]* -> *[c*sy, d*sy, a*sx, b*sx, e, f]*.

Parameters

- ***sx*** (*float*) – Shearing effect in X direction. See attribute *c*.
- ***sy*** (*float*) – Shearing effect in Y direction. See attribute *b*.

`preTranslate` (*tx, ty*)

Modify the matrix to perform a shifting / translation operation along the x and / or y axis. Has effects on attributes *e* and *f* only: *[a, b, c, d, e, f]* -> *[a, b, c, d, tx*a + ty*c, tx*b + ty*d]*.

Parameters

- ***tx*** (*float*) – Translation effect in X direction. See attribute *e*.
- ***ty*** (*float*) – Translation effect in Y direction. See attribute *f*.

`concat` (*m1, m2*)

Calculate the matrix product *m1 * m2* and store the result in the current matrix. Any of *m1* or *m2* may be

the current matrix. Be aware that matrix multiplication is not commutative. So the sequence of $m1$, $m2$ is important.

Parameters

- **m1** (*Matrix*) – First (left) matrix.
- **m2** (*Matrix*) – Second (right) matrix.

invert ($m = None$)

Calculate the matrix inverse of m and store the result in the current matrix. Returns 1 if m is not invertible (“degenerate”). In this case the current matrix **will not change**. Returns 0 if m is invertible, and the current matrix is replaced with the inverted m .

Parameters **m** (*Matrix*) – Matrix to be inverted. If not provided, the current matrix will be used.

Return type int

a

Scaling in X-direction (**width**). For example, a value of 0.5 performs a shrink of the **width** by a factor of 2. If $a < 0$, a left-right flip will (additionally) occur.

Type float

b

Causes a shearing effect: each $Point(x, y)$ will become $Point(x, y - b*x)$. Therefore, looking from left to right, e.g. horizontal lines will be “tilt” – downwards if $b > 0$, upwards otherwise (b is the tangens of the tilting angle).

Type float

c

Causes a shearing effect: each $Point(x, y)$ will become $Point(x - c*y, y)$. Therefore, looking upwards, vertical lines will be “tilt” – to the left if $c > 0$, to the right otherwise (c is the tangens of the tilting angle).

Type float

d

Scaling in Y-direction (**height**). For example, a value of 1.5 performs a stretch of the **height** by 50%. If $d < 0$, an up-down flip will (additionally) occur.

Type float

e

Causes a horizontal shift effect: Each $Point(x, y)$ will become $Point(x + e, y)$. Positive (negative) values of e will shift right (left).

Type float

f

Causes a vertical shift effect: Each $Point(x, y)$ will become $Point(x, y - f)$. Positive (negative) values of f will shift down (up).

Type float

isRectilinear

Rectilinear means that no shearing is present and that any rotations are integer multiples of 90 degrees. Usually this is used to confirm that (axis-aligned) rectangles before the transformation are still axis-aligned rectangles afterwards.

Type bool

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
 - A matrix can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.
 - Changes of matrix properties and execution of matrix methods can be executed consecutively. This is the same as multiplying the respective matrices.
 - Matrix multiplication is **not commutative** – changing the execution sequence in general changes the result. So it can quickly become unclear which result a transformation will yield.
-

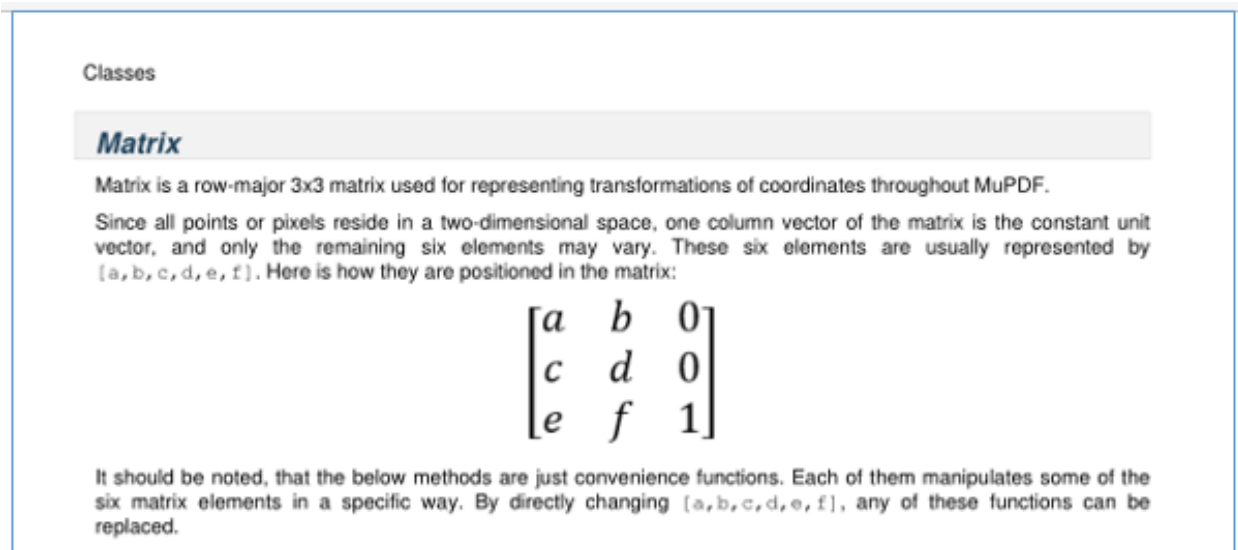
To keep results foreseeable for a series of matrix operations, Adobe recommends the following approach (*Adobe PDF References*, page 206):

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

6.10.1 Examples

Here are examples to illustrate some of the effects achievable. The following pictures start with a page of the PDF version of this help file. We show what happens when a matrix is being applied (though always full pages are created, only parts are displayed here to save space).

This is the original page image:

The image is a screenshot of the PyMuPDF documentation page for the Matrix class. It shows the class name 'Matrix' in a blue header, followed by a description: '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:'. Below this is a 3x3 matrix representation:
$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$
 At the bottom of the screenshot, it says: '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.'

6.10.2 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 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:

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

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

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:

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

6.10.3 Flipping

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

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, d, c, b, e, f]`. Here is how they are positioned in the matrix:

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

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

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, d, c, b, e, f]`. Here is how they are positioned in the matrix:

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

6.10.4 Shearing

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

Classes

Matrix

Matrix is a row-major 3x3 matrix used image transformations in MuPDF. With matrices you can manipulate the rendered image of a page in a variety of ways: (parts of) pages can be rotated, zoomed, flipped, sheared and shifted by setting some or all of just six numerical values.

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

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

It should be noted, that

- the below methods are just convenience functions. Even manipulating $[a, b, c, d, e, f]$
- all manipulations can be combined - you can one

Methods

Matrix.
Matrix.

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

Classes

Matrix

Matrix is a row-major 3x3 matrix used image transformations in MuPDF. With matrices you can manipulate the rendered image of a page in a variety of ways: (parts of) pages can be rotated, zoomed, flipped, sheared and shifted by setting some or all of just six numerical values.

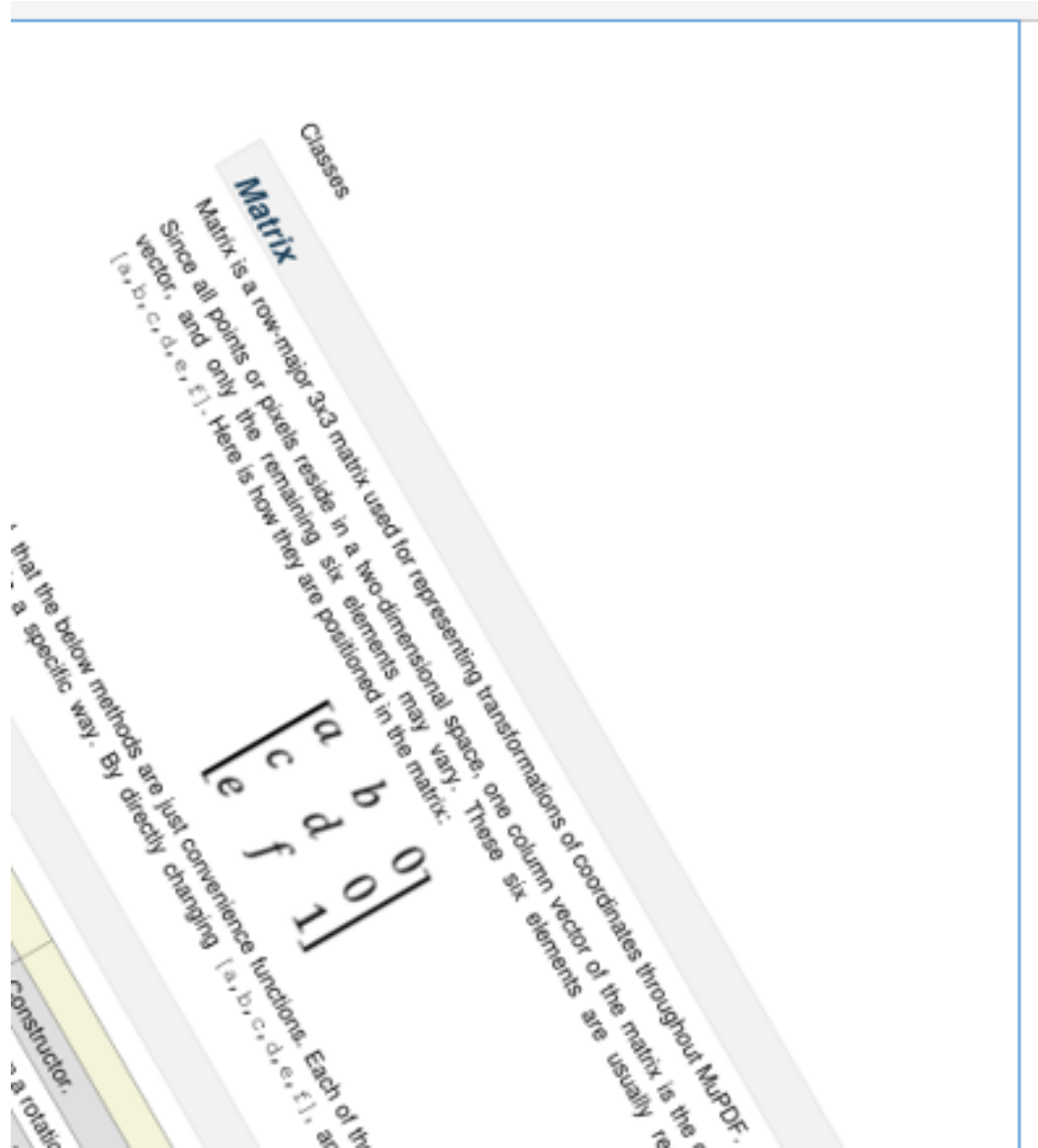
Since all points or pixels live in a two-dimensional space, one column vector of that matrix is a constant unit vector, and only the remaining six elements are used for manipulations. These six elements are usually represented by $[a, b, c, d, e, f]$. Here is how they are positioned in the matrix:

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

It should be noted, that

6.10.5 Rotating

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



6.11 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
<code>Outline.down</code>	next item downwards
<code>Outline.next</code>	next item same level
<code>Outline.page</code>	page number (0-based)
<code>Outline.title</code>	title
<code>Outline.uri</code>	string further specifying the outline target
<code>Outline.isExternal</code>	target is outside this document
<code>Outline.is_open</code>	whether sub-outlines are open or collapsed
<code>Outline.isOpen</code>	whether sub-outlines are open or collapsed
<code>Outline.dest</code>	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*

6.12 Page

Class representing a document page. A page object is created by `Document.loadPage()` or, equivalently, via indexing the document like `doc[n]` - it has no independent constructor.

There is a parent-child relationship between a document and its pages. If the document is closed or deleted, all page objects (and their respective children, too) in existence will become unusable (“orphaned”): If a page property or method is being used, an exception is raised.

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

6.12.1 Modifying Pages

Changing page properties and adding or changing page content is available for PDF documents only.

In a nutshell, this is what you can do with PyMuPDF:

- Modify page rotation and the visible part (“CropBox”) of the page.
- Insert images, other PDF pages, text and simple geometrical objects.
- Add annotations and form fields.

Note: Methods require coordinates (points, rectangles) to put content in desired places. Please be aware that since v1.17.0 these coordinates **must always** be provided relative to the **unrotated** page. The reverse is also true: except `Page.rect`, resp. `Page.bound()` (both *reflect* when the page is rotated), all coordinates returned by methods and attributes pertain to the unrotated page.

So the returned value of e.g. `Page.getImageBbox()` will not change if you do a `Page.setRotation()`. The same is true for coordinates returned by `Page.getText()`, annotation rectangles, and so on. If you want to find out, where an object is located in **rotated coordinates**, multiply the coordinates with `Page.rotationMatrix`. There also is its inverse, `Page.derotationMatrix`, which you can use when interfacing with other readers, which may behave differently in this respect.

Note: If you add or update annotations, links or form fields on the page and immediately afterwards need to work with them (i.e. **without leaving the page**), you should reload the page using `Document.reload_page()` before referring to these new or updated items.

This ensures all your changes have been fully applied to PDF structures, so can safely create Pixmaps or successfully iterate over annotations, links and form fields.

Method / Attribute	Short Description
<code>Page.addCaretAnnot()</code>	PDF only: add a caret annotation
<code>Page.addCircleAnnot()</code>	PDF only: add a circle annotation
<code>Page.addFileAnnot()</code>	PDF only: add a file attachment annotation
<code>Page.addFreetextAnnot()</code>	PDF only: add a text annotation
<code>Page.addHighlightAnnot()</code>	PDF only: add a “highlight” annotation
<code>Page.addInkAnnot()</code>	PDF only: add an ink annotation
<code>Page.addLineAnnot()</code>	PDF only: add a line annotation
<code>Page.addPolygonAnnot()</code>	PDF only: add a polygon annotation
<code>Page.addPolylineAnnot()</code>	PDF only: add a multi-line annotation
<code>Page.addRectAnnot()</code>	PDF only: add a rectangle annotation

Continued on next page

Table 3 – continued from previous page

Method / Attribute	Short Description
<code>Page.addRedactAnnot ()</code>	PDF only: add a redaction annotation
<code>Page.addSquigglyAnnot ()</code>	PDF only: add a “squiggly” annotation
<code>Page.addStampAnnot ()</code>	PDF only: add a “rubber stamp” annotation
<code>Page.addStrikeoutAnnot ()</code>	PDF only: add a “strike-out” annotation
<code>Page.addTextAnnot ()</code>	PDF only: add a comment
<code>Page.addUnderlineAnnot ()</code>	PDF only: add an “underline” annotation
<code>Page.addWidget ()</code>	PDF only: add a PDF Form field
<code>Page.annot_names ()</code>	PDF only: a list of annotation and widget names
<code>Page.annots ()</code>	return a generator over the annots on the page
<code>Page.apply_redactions ()</code>	PDF only: process the redactions of the page
<code>Page.bound ()</code>	rectangle of the page
<code>Page.deleteAnnot ()</code>	PDF only: delete an annotation
<code>Page.deleteWidget ()</code>	PDF only: delete a widget / field
<code>Page.deleteLink ()</code>	PDF only: delete a link
<code>Page.drawBezier ()</code>	PDF only: draw a cubic Bezier curve
<code>Page.drawCircle ()</code>	PDF only: draw a circle
<code>Page.drawCurve ()</code>	PDF only: draw a special Bezier curve
<code>Page.drawLine ()</code>	PDF only: draw a line
<code>Page.drawOval ()</code>	PDF only: draw an oval / ellipse
<code>Page.drawPolyline ()</code>	PDF only: connect a point sequence
<code>Page.drawRect ()</code>	PDF only: draw a rectangle
<code>Page.drawSector ()</code>	PDF only: draw a circular sector
<code>Page.drawSquiggle ()</code>	PDF only: draw a squiggly line
<code>Page.drawZigzag ()</code>	PDF only: draw a zig-zagged line
<code>Page.getDrawings ()</code>	get list of the draw commands contained in the page
<code>Page.getFontList ()</code>	PDF only: get list of used fonts
<code>Page.getImageBbox ()</code>	PDF only: get bbox of embedded image
<code>Page.getImageList ()</code>	PDF only: get list of used images
<code>Page.getLinks ()</code>	get all links
<code>Page.getPixmap ()</code>	create a page image in raster format
<code>Page.getSVGImage ()</code>	create a page image in SVG format
<code>Page.getText ()</code>	extract the page’s text
<code>Page.getTextbox ()</code>	extract text contained in a rectangle
<code>Page.getTextPage ()</code>	create a TextPage for the page
<code>Page.insertFont ()</code>	PDF only: insert a font for use by the page
<code>Page.insertImage ()</code>	PDF only: insert an image
<code>Page.insertLink ()</code>	PDF only: insert a link
<code>Page.insertText ()</code>	PDF only: insert text
<code>Page.insertTextbox ()</code>	PDF only: insert a text box
<code>Page.links ()</code>	return a generator of the links on the page
<code>Page.loadAnnot ()</code>	PDF only: load a specific annotation
<code>Page.loadLinks ()</code>	return the first link on a page
<code>Page.newShape ()</code>	PDF only: create a new <i>Shape</i>
<code>Page.searchFor ()</code>	search for a string
<code>Page.setCropBox ()</code>	PDF only: modify the visible page
<code>Page.setMediaBox ()</code>	PDF only: modify the mediabox
<code>Page.setRotation ()</code>	PDF only: set page rotation
<code>Page.showPDFpage ()</code>	PDF only: display PDF page image
<code>Page.updateLink ()</code>	PDF only: modify a link

Continued on next page

Table 3 – continued from previous page

Method / Attribute	Short Description
<code>Page.widgets()</code>	return a generator over the fields on the page
<code>Page.writeText()</code>	write one or more <i>TextWriter</i> objects
<code>Page.CropBox</code>	the page's <i>CropBox</i>
<code>Page.CropBoxPosition</code>	displacement of the <i>CropBox</i>
<code>Page.firstAnnot</code>	first <i>Annot</i> on the page
<code>Page.firstLink</code>	first <i>Link</i> on the page
<code>Page.firstWidget</code>	first widget (form field) on the page
<code>Page.MediaBox</code>	the page's <i>MediaBox</i>
<code>Page.MediaBoxSize</code>	bottom-right point of <i>MediaBox</i>
<code>Page.derotationMatrix</code>	PDF only: get coordinates in unrotated page space
<code>Page.rotationMatrix</code>	PDF only: get coordinates in rotated page space
<code>Page.transformationMatrix</code>	PDF only: translate between PDF and MuPDF space
<code>Page.number</code>	page number
<code>Page.parent</code>	owning document object
<code>Page.rect</code>	rectangle of the page
<code>Page.rotation</code>	PDF only: page rotation
<code>Page.xref</code>	PDF only: page <i>xref</i>

Class API

class Page

`bound()`

Determine the rectangle of the page. Same as property `Page.rect` below. For PDF documents this **usually** also coincides with `MediaBox` and `CropBox`, but not always. For example, if the page is rotated, then this is reflected by this method – the `Page.CropBox` however will not change.

Return type *Rect*

`addCaretAnnot(point)`

(New in version 1.16.0)

PDF only: Add a caret icon. A caret annotation is a visual symbol normally used to indicate the presence of text edits on the page.

Parameters `point` (*point_like*) – the top left point of a 20 x 20 rectangle containing the MuPDF-provided icon.

Return type *Annot*

Returns the created annotation.



`addTextAnnot(point, text, icon="Note")`

PDF only: Add a comment icon (“sticky note”) with accompanying text. Only the icon is visible, the accompanying text is hidden and can be visualized by many PDF viewers by hovering the mouse over the symbol.

Parameters

- `point` (*point_like*) – the top left point of a 20 x 20 rectangle containing the MuPDF-provided “note” icon.

- **text** (*str*) – the commentary text. This will be shown on double clicking or hovering over the icon. May contain any Latin characters.
- **icon** (*str*) – (*new in version 1.16.0*) choose one of “Note” (default), “Comment”, “Help”, “Insert”, “Key”, “NewParagraph”, “Paragraph” as the visual symbol for the embodied text⁴.

Return type *Annot*

Returns the created annotation.

addFreetextAnnot (*rect*, *text*, *fontsize=12*, *fontname="helv"*, *text_color=0*, *fill_color=1*, *rotate=0*, *align=TEXT_ALIGN_LEFT*)

PDF only: Add text in a given rectangle.

Parameters

- **rect** (*rect_like*) – the rectangle into which the text should be inserted. Text is automatically wrapped to a new line at box width. Lines not fitting into the box will be invisible.
- **text** (*str*) – the text. (*New in v1.17.0*) May contain any mixture of Latin, Greek, Cyrillic, Chinese, Japanese and Korean characters. The respective required font is automatically determined.
- **fontsize** (*float*) – the font size. Default is 12.
- **fontname** (*str*) – the font name. Default is “Helv”. Accepted alternatives are “Cour”, “TiRo”, “ZaDb” and “Symb”. The name may be abbreviated to the first two characters, like “Co” for “Cour”. Lower case is also accepted. (*Changed in v1.16.0*) Bold or italic variants of the fonts are **no longer accepted**. A user-contributed script provides a circumvention for this restriction – see section *Using Buttons and JavaScript* in chapter *Collection of Recipes*. (*New in v1.17.0*) The actual font to use is now determined on a by-character level, and all required fonts (or sub-fonts) are automatically included. Therefore, you should rarely ever need to care about this parameter and let it default (except you insist on a serified font for your non-CJK text parts).
- **text_color** (*sequence*, *float*) – (*new in version 1.16.0*) the text color. Default is black.
- **fill_color** (*sequence*, *float*) – (*new in version 1.16.0*) the fill color. Default is white.
- **align** (*int*) – (*new in version 1.17.0*) text alignment, one of TEXT_ALIGN_LEFT, TEXT_ALIGN_CENTER, TEXT_ALIGN_RIGHT - justify is not supported.
- **rotate** (*int*) – the text orientation. Accepted values are 0, 90, 270, invalid entries are set to zero.

Return type *Annot*

Returns the created annotation. Color properties **can only be changed** using special parameters of *Annot.update()*. There, you can also set a border color different from the text color.

addFileAnnot (*pos*, *buffer*, *filename*, *ufilename=None*, *desc=None*, *icon="PushPin"*)

PDF only: Add a file attachment annotation with a “PushPin” icon at the specified location.

Parameters

⁴ You are generally free to choose any of the *Annotation Icons in MuPDF* you consider adequate.

- **pos** (*point_like*) – the top-left point of a 18x18 rectangle containing the MuPDF-provided “PushPin” icon.
- **buffer** (*bytes, bytearray, BytesIO*) – the data to be stored (actual file content, any data, etc.).
Changed in version 1.14.13 *io.BytesIO* is now also supported.
- **filename** (*str*) – the filename to associate with the data.
- **ufilename** (*str*) – the optional PDF unicode version of filename. Defaults to filename.
- **desc** (*str*) – an optional description of the file. Defaults to filename.
- **icon** (*str*) – (*new in version 1.16.0*) choose one of “PushPin” (default), “Graph”, “Paperclip”, “Tag” as the visual symbol for the attached data⁴.

Return type *Annot*

Returns the created annotation. Use methods of *Annot* to make any changes.

addInkAnnot (*list*)

PDF only: Add a “freehand” scribble annotation.

Parameters **list** (*sequence*) – a list of one or more lists, each containing *point_like* items. Each item in these sublists is interpreted as a *Point* through which a connecting line is drawn. Separate sublists thus represent separate drawing lines.

Return type *Annot*

Returns the created annotation in default appearance (black line of width 1). Use annotation methods with a subsequent *Annot.update()* to modify.

addLineAnnot (*p1, p2*)

PDF only: Add a line annotation.

Parameters

- **p1** (*point_like*) – the starting point of the line.
- **p2** (*point_like*) – the end point of the line.

Return type *Annot*

Returns the created annotation. It is drawn with line color black and line width 1. The **rectangle** is automatically created to contain both points, each one surrounded by a circle of radius 3 * line width to make room for any line end symbols.

addRectAnnot (*rect*)

addCircleAnnot (*rect*)

PDF only: Add a rectangle, resp. circle annotation.

Parameters **rect** (*rect_like*) – the rectangle in which the circle or rectangle is drawn, must be finite and not empty. If the rectangle is not equal-sided, an ellipse is drawn.

Return type *Annot*

Returns the created annotation. It is drawn with line color red, no fill color and line width 1.

addRedactAnnot (*quad, text=None, fontname=None, fontsize=11, align=TEXT_ALIGN_LEFT, fill=(1, 1, 1), text_color=(0, 0, 0), cross_out=True*)

PDF only: (*new in version 1.16.11*) Add a redaction annotation. A redaction annotation identifies content to be removed from the document. Adding such an annotation is the first of two steps. It makes visible what will be removed in the subsequent step, *Page.apply_redactions()*.

Parameters

- **quad** (*quad_like*, *rect_like*) – specifies the (rectangular) area to be removed which is always equal to the annotation rectangle. This may be a *rect_like* or *quad_like* object. If a quad is specified, then the enveloping rectangle is taken.
- **text** (*str*) – (New in v1.16.12) text to be placed in the rectangle after applying the redaction (and thus removing old content).
- **fontname** (*str*) – (New in v1.16.12) the font to use when *text* is given, otherwise ignored. The same rules apply as for *Page.insertTextbox()* – which is the method *Page.apply_redactions()* internally invokes. The replacement text will be **vertically centered**, if this is one of the CJK or *PDF Base 14 Fonts*.

Note:

- For an **existing** font of the page, use its reference name as *fontname* (this is *item[4]* of its entry in *Page.getFontList()*).
- For a **new, non-builtin** font, proceed as follows:

```

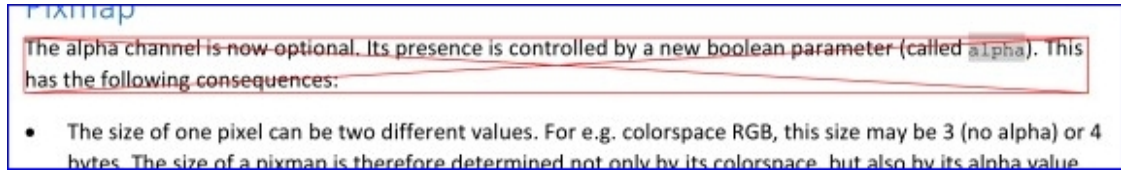
page.insertText(point, # anywhere, but outside all_
↳redaction rectangles
    "something", # some non-empty string
    fontname="newname", # new, unused reference name
    fontfile="...", # desired font file
    render_mode=3, # makes the text invisible
)
page.addRedactAnnot(..., fontname="newname")

```

- **fontsize** (*float*) – (New in v1.16.12) the fontsize to use for the replacing text. If the text is too large to fit, several insertion attempts will be made, gradually reducing the fontsize to no less than 4. If then the text will still not fit, no text insertion will take place at all.
- **align** (*int*) – (New in v1.16.12) the horizontal alignment for the replacing text. See *insertTextbox()* for available values. The vertical alignment is (approximately) centered if a PDF built-in font is used (CJK or *PDF Base 14 Fonts*).
- **fill** (*sequence*) – (New in v1.16.12) the fill color of the rectangle **after applying** the redaction. The default is *white = (1, 1, 1)*, which is also taken if *None* is specified. (Changed in v1.16.13) To suppress a fill color altogether, specify *False*. In this cases the rectangle remains transparent.
- **text_color** (*sequence*) – (New in v1.16.12) the color of the replacing text. Default is *black = (0, 0, 0)*.
- **cross_out** (*bool*) – (new in v1.17.2) add two diagonal lines to the annotation rectangle.

Return type *Annot*

Returns the created annotation. (Changed in v1.17.2) Its standard appearance looks like a red rectangle (no fill color), optionally showing two diagonal lines. Colors, line width, dashing, opacity and blend mode can now be set and applied via *Annot.update()* like with other annotations.



addPolylineAnnot (*points*)

addPolygonAnnot (*points*)

PDF only: Add an annotation consisting of lines which connect the given points. A **Polygon's** first and last points are automatically connected, which does not happen for a **PolyLine**. The **rectangle** is automatically created as the smallest rectangle containing the points, each one surrounded by a circle of radius 3 (= 3 * line width). The following shows a 'PolyLine' that has been modified with colors and line ends.

Parameters *points* (*list*) – a list of *point_like* objects.

Return type *Annot*

Returns the created annotation. It is drawn with line color black, no fill color and line width 1. Use methods of *Annot* to make any changes to achieve something like this:



addUnderlineAnnot (*quads=None, start=None, stop=None, clip=None*)

addStrikeoutAnnot (*quads=None, start=None, stop=None, clip=None*)

addSquigglyAnnot (*quads=None, start=None, stop=None, clip=None*)

addHighlightAnnot (*quads=None, start=None, stop=None, clip=None*)

PDF only: These annotations are normally used for **marking text** which has previously been somehow located (for example via *Page.searchFor()*). But this is not required: you are free to “mark” just anything.

Standard colors are chosen per annotation type: **yellow** for highlighting, **red** for strike out, **green** for underlining, and **magenta** for wavy underlining.

The methods convert the arguments into a list of *Quad* objects. The **annotation** rectangle is then calculated to envelop all these quadrilaterals.

Note: *searchFor()* delivers a list of either rectangles or quadrilaterals. Such a list can be directly used as parameter for these annotation types and will deliver **one common** annotation for all occurrences of the search string:

```
>>> quads = page.searchFor("pymupdf", quads=True)
>>> page.addHighlightAnnot(quads)
```

Parameters

- **quads** (*rect_like, quad_like, list, tuple*) – (Changed in v1.14.20) the location(s) – rectangle(s) or quad(s) – to be marked. A list or tuple must consist of *rect_like* or *quad_like* items (or even a mixture of either). Every item must be finite, convex and not empty (as applicable). (Changed in v1.16.14) **Set this parameter to None** if you want to use the following arguments.

- **start** (*point_like*) – (New in v1.16.14) start text marking at this point. Defaults to the top-left point of *clip*.
- **stop** (*point_like*) – (New in v1.16.14) stop text marking at this point. Defaults to the bottom-right point of *clip*.
- **clip** (*rect_like*) – (New in v1.16.14) only consider text lines intersecting this area. Defaults to the page rectangle.

Return type *Annot* or (changed in v1.16.14) *None*

Returns the created annotation. (Changed in v1.16.14) If *quads* is an empty list, **no annotation** is created. To change colors, set the “stroke” color accordingly (`Annot.setColors()`) and then perform an `Annot.update()`.

Note: Starting with v1.16.14 you can use parameters *start*, *stop* and *clip* to highlight consecutive lines between the points *start* and *stop*. Make use of *clip* to further reduce the selected line bboxes and thus deal with e.g. multi-column pages. The following multi-line highlight on a page with three text columns was created by specifying the two red points and setting *clip* accordingly.



addStampAnnot (*rect*, *stamp=0*)

PDF only: Add a “rubber stamp” like annotation to e.g. indicate the document’s intended use (“DRAFT”, “CONFIDENTIAL”, etc.).

Parameters

- **rect** (*rect_like*) – rectangle where to place the annotation.
- **stamp** (*int*) – id number of the stamp text. For available stamps see *Stamp Annotation Icons*.

Note:

- The stamp’s text and its border line will automatically be sized and be put horizontally and vertically centered in the given rectangle. `Annot.rect` is automatically calculated to fit the given **width** and will usually be smaller than this parameter.
- The font chosen is “Times Bold” and the text will be upper case.
- The appearance can be changed using `Annot.setOpacity()` and by setting the “stroke” color (no “fill” color supported).
- This can be used to create watermark images: on a temporary PDF page create a stamp annotation with a low opacity value, make a pixmap from it with `alpha=True` (and potentially also rotate it), discard the temporary PDF page and use the pixmap with `insertImage()` for your target PDF.

NOT FOR
PUBLIC RELEASE

'Stamp' annotation

addWidget (*widget*)

PDF only: Add a PDF Form field (“widget”) to a page. This also **turns the PDF into a Form PDF**. Because of the large amount of different options available for widgets, we have developed a new class *Widget*, which contains the possible PDF field attributes. It must be used for both, form field creation and updates.

Parameters *widget* (*Widget*) – a *Widget* object which must have been created upfront.

Returns a widget annotation.

deleteAnnot (*annot*)

PDF only: Delete annotation from the page and return the next one.

Changed in version 1.16.6 The removal will now include any bound ‘Popup’ or response annotations and related objects.

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

Return type *Annot*

Returns the annotation following the deleted one. Please remember that physical removal requires saving to a new file with garbage > 0.

deleteWidget (*widget*)

(New in v1.18.4)

PDF only: Delete field from the page and return the next one.

Parameters *widget* (*Widget*) – the widget to be deleted.

Return type *Widget*

Returns the widget following the deleted one. Please remember that physical removal requires saving to a new file with garbage > 0.

apply_redactions (*images=PDF_REDACT_IMAGE_PIXELS*)

(New in version 1.16.11)

PDF only: Remove all **text content** contained in any redaction rectangle.

(Changed in v1.16.12) The previous *mark* parameter is gone. Instead, the respective rectangles are filled with the individual *fill* color of each redaction annotation. If a *text* was given in the annotation, then *insertTextbox()* is invoked to insert it, using parameters provided with the redaction.

This method applies and then deletes all redactions from the page.

Parameters *images* (*int*) – (new in v1.18.0) how to redact overlapping images. The default (2) blanks out overlapping pixels. *PDF_REDACT_IMAGE_NONE* (0) ignores, and *PDF_REDACT_IMAGE_REMOVE* (1) completely removes all overlapping images.

Returns *True* if at least one redaction annotation has been processed, *False* otherwise.

Note: Text contained in a redaction rectangle will be **physically** removed from the page and will no longer appear in e.g. text extractions or anywhere else. Other annotations are unaffected.

All overlapping links will be removed.

(*Changed in v1.18.0*) The overlapping parts of **images** will be blanked-out for option 2. Option 0 does not touch any images and 1 will remove any image with an overlap.

Please be aware that there is a MuPDF bug for option `PDF_REDACT_IMAGE_PIXELS = 2`: transparent images will be incorrectly handled!

To remove only selected images (as opposed to all intersecting), use PyMuPDF low-level functions instead of redaction annotations.

Text removal is done by character: A character is removed if its bbox has a **non-empty intersection** with a redaction (*changed in MuPDF v1.17*).

Redactions are an easy way to replace single words in a PDF, or to just physically remove them from the PDF: locate the word “secret” using some text extraction or search method and insert a redaction using “xxxxxx” as replacement text for each occurrence.

- Be wary if the replacement is longer than the original – this may lead to an awkward appearance, line breaks or no new text at all.
- For a number of reasons, the new text may not exactly be positioned on the same line like the old one – especially true if the replacement font was not one of CJK or *PDF Base 14 Fonts*.

deleteLink (*linkdict*)

PDF only: Delete the specified link from the page. The parameter must be an **original item** of `getLinks()` (see below). The reason for this is the dictionary’s “*xref*” key, which identifies the PDF object to be deleted.

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

insertLink (*linkdict*)

PDF only: Insert a new link on this page. The parameter must be a dictionary of format as provided by `getLinks()` (see below).

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

updateLink (*linkdict*)

PDF only: Modify the specified link. The parameter must be a (modified) **original item** of `getLinks()` (see below). The reason for this is the dictionary’s “*xref*” key, which identifies the PDF object to be changed.

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

getLinks ()

Retrieves **all** links of a page.

Return type list

Returns A list of dictionaries. For a description of the dictionary entries see below. Always use this or the `Page.links()` method if you intend to make changes to the links of a page.

links (*kinds=None*)

(*New in version 1.16.4*)

Return a generator over the page’s links. The results equal the entries of `Page.getLinks()`.

Parameters *kinds* (*sequence*) – a sequence of integers to down-select to one or more link kinds. Default is all links. Example: `kinds=(fitz.LINK_GOTO,)` will only return internal links.

Return type generator

Returns an entry of `Page.getLinks()` for each iteration.

annots (*types=None*)
(New in version 1.16.4)

Return a generator over the page's annotations.

Parameters **types** (*sequence*) – a sequence of integers to down-select to one or annotation types. Default is all annotations. Example: *types=(fitz.PDF_ANNOT_FREETEXT, fitz.PDF_ANNOT_TEXT)* will only return 'FreeText' and 'Text' annotations.

Return type generator

Returns an *Annot* for each iteration.

widgets (*types=None*)
(New in version 1.16.4)

Return a generator over the page's form fields.

Parameters **types** (*sequence*) – a sequence of integers to down-select to one or more widget types. Default is all form fields. Example: *types=(fitz.PDF_WIDGET_TYPE_TEXT,)* will only return 'Text' fields.

Return type generator

Returns a *Widget* for each iteration.

writeText (*rect=None, writers=None, overlay=True, color=None, opacity=None, keep_proportion=True, rotate=0, oc=0*)
(New in version 1.16.18)

PDF only: Write the text of one or more *TextWriter* objects to the page.

Parameters

- **rect** (*rect_like*) – where to place the text. If omitted, the rectangle union of the text writers is used.
- **writers** (*sequence*) – a non-empty tuple / list of *TextWriter* objects or a single *TextWriter*.
- **opacity** (*float*) – set transparency, overwrites resp. value in the text writers.
- **color** (*sequ*) – set the text color, overwrites resp. value in the text writers.
- **overlay** (*bool*) – put the text in foreground or background.
- **keep_proportion** (*bool*) – maintain the aspect ratio.
- **rotate** (*float*) – rotate the text by an arbitrary angle.
- **oc** (*int*) – (new in v1.18.4) the *xref* of an *OCG* or *OCMD*.

Note: Parameters *overlay*, *keep_proportion*, *rotate* and *oc* have the same meaning as in *showPDFpage*.

insertText (*point, text, fontsize=11, fontname="helv", fontfile=None, idx=0, color=None, fill=None, render_mode=0, border_width=1, encoding=TEXT_ENCODING_LATIN, rotate=0, morph=None, stroke_opacity=1, fill_opaity=1, overlay=True, oc=0*)
(Changed in v1.18.4)

PDF only: Insert text starting at *point_like point*. See *Shape.insertText()*.

insertTextbox (*rect*, *buffer*, *fontsize=11*, *fontname="helv"*, *fontfile=None*, *idx=0*, *color=None*, *fill=None*, *render_mode=0*, *border_width=1*, *encoding=TEXT_ENCODING_LATIN*, *expandtabs=8*, *align=TEXT_ALIGN_LEFT*, *charwidths=None*, *rotate=0*, *morph=None*, *stroke_opacity=1*, *fill_opaity=1*, *oc=0*, *overlay=True*)

(Changed in v1.18.4)

PDF only: Insert text into the specified *rect_like* *rect*. See *Shape.insertTextbox()*.

drawLine (*p1*, *p2*, *color=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opaity=1*, *oc=0*)

(Changed in v1.18.4)

PDF only: Draw a line from *p1* to *p2* (*point_like* s). See *Shape.drawLine()*.

drawZigzag (*p1*, *p2*, *breadth=2*, *color=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opaity=1*, *oc=0*)

(Changed in v1.18.4)

PDF only: Draw a zigzag line from *p1* to *p2* (*point_like* s). See *Shape.drawZigzag()*.

drawSquiggle (*p1*, *p2*, *breadth=2*, *color=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opaity=1*, *oc=0*)

(Changed in v1.18.4)

PDF only: Draw a squiggly (wavy, undulated) line from *p1* to *p2* (*point_like* s). See *Shape.drawSquiggle()*.

drawCircle (*center*, *radius*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opaity=1*, *oc=0*)

(Changed in v1.18.4)

PDF only: Draw a circle around *center* (*point_like*) with a radius of *radius*. See *Shape.drawCircle()*.

drawOval (*quad*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opaity=1*, *oc=0*)

(Changed in v1.18.4)

PDF only: Draw an oval (ellipse) within the given *rect_like* or *quad_like*. See *Shape.drawOval()*.

drawSector (*center*, *point*, *angle*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *fullSector=True*, *overlay=True*, *closePath=False*, *morph=None*, *stroke_opacity=1*, *fill_opaity=1*, *oc=0*)

(Changed in v1.18.4)

PDF only: Draw a circular sector, optionally connecting the arc to the circle's center (like a piece of pie). See *Shape.drawSector()*.

drawPolyline (*points*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *closePath=False*, *morph=None*, *stroke_opacity=1*, *fill_opaity=1*, *oc=0*)

(Changed in v1.18.4)

PDF only: Draw several connected lines defined by a sequence of *point_like* s. See *Shape.drawPolyline()*.

drawBezier (*p1*, *p2*, *p3*, *p4*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *closePath=False*, *morph=None*, *stroke_opacity=1*, *fill_opaity=1*, *oc=0*)

(Changed in v1.18.4)

PDF only: Draw a cubic Bézier curve from *p1* to *p4* with the control points *p2* and *p3* (all are *'data_point_like'* s). See *Shape.drawBezier()*.

drawCurve (*p1, p2, p3, color=None, fill=None, width=1, dashes=None, lineCap=0, lineJoin=0, overlay=True, closePath=False, morph=None, stroke_opacity=1, fill_opacity=1, oc=0*)
(Changed in v1.18.4)

PDF only: This is a special case of *drawBezier()*. See *Shape.drawCurve()*.

drawRect (*rect, color=None, fill=None, width=1, dashes=None, lineCap=0, lineJoin=0, overlay=True, morph=None, stroke_opacity=1, fill_opacity=1, oc=0*)
(Changed in v1.18.4)

PDF only: Draw a rectangle. See *Shape.drawRect()*.

Note: An efficient way to background-color a PDF page with the old Python paper color is

```
>>> col = fitz.utils.getColor("py_color")
>>> page.drawRect(page.rect, color=col, fill=col, overlay=False)
```

insertFont (*fontname="helv", fontfile=None, fontbuffer=None, set_simple=False, encoding=TEXT_ENCODING_LATIN*)

PDF only: Add a new font to be used by text output methods and return its *xref*. If not already present in the file, the font definition will be added. Supported are the built-in *Base14_Fonts* and the CJK fonts via “reserved” fontnames. Fonts can also be provided as a file path or a memory area containing the image of a font file.

Parameters *fontname* (*str*) – The name by which this font shall be referenced when outputting text on this page. In general, you have a “free” choice here (but consult the *Adobe PDF References*, page 56, section 3.2.4 for a formal description of building legal PDF names). However, if it matches one of the *Base14_Fonts* or one of the CJK fonts, *fontfile* and *fontbuffer* are ignored.

In other words, you cannot insert a font via *fontfile* / *fontbuffer* and also give it a reserved *fontname*.

Note: A reserved fontname can be specified in any mixture of upper or lower case and still match the right built-in font definition: fontnames “helv”, “Helv”, “HELV”, “Helvetica”, etc. all lead to the same font definition “Helvetica”. But from a *Page* perspective, these are **different references**. You can exploit this fact when using different *encoding* variants (Latin, Greek, Cyrillic) of the same font on a page.

Parameters

- **fontfile** (*str*) – a path to a font file. If used, *fontname* must be **different from all reserved names**.
- **fontbuffer** (*bytes/bytearray*) – the memory image of a font file. If used, *fontname* must be **different from all reserved names**. This parameter would typically be used to transfer fonts between different pages of the same or different PDFs.
- **set_simple** (*int*) – applicable for *fontfile* / *fontbuffer* cases only: enforce treatment as a “simple” font, i.e. one that only uses character codes up to 255.
- **encoding** (*int*) – applicable for the “Helvetica”, “Courier” and “Times” sets of *Base14_Fonts* only. Select one of the available encodings Latin (0), Cyrillic (2) or Greek (1). Only use the default (0 = Latin) for “Symbol” and “ZapfDingBats”.

Rytp *int*

Returns the *xref* of the installed font.

Note: Built-in fonts will not lead to the inclusion of a font file. So the resulting PDF file will remain small. However, your PDF viewer software is responsible for generating an appropriate appearance – and there **exist** differences on whether or how each one of them does this. This is especially true for the CJK fonts. But also Symbol and ZapfDingbats are incorrectly handled in some cases. Following are the **Font Names** and their correspondingly installed **Base Font** names:

Base-14 Fonts¹

Font Name	Installed Base Font	Comments
helv	Helvetica	normal
heit	Helvetica-Oblique	italic
hebo	Helvetica-Bold	bold
hebi	Helvetica-BoldOblique	bold-italic
cour	Courier	normal
coit	Courier-Oblique	italic
cobo	Courier-Bold	bold
cobi	Courier-BoldOblique	bold-italic
tiro	Times-Roman	normal
tiit	Times-Italic	italic
tibo	Times-Bold	bold
tibi	Times-BoldItalic	bold-italic
symb	Symbol	³
zadb	ZapfDingbats	³

CJK Fonts² (China, Japan, Korea)

Font Name	Installed Base Font	Comments
china-s	Heiti	simplified Chinese
china-ss	Song	simplified Chinese (serif)
china-t	Fangti	traditional Chinese
china-ts	Ming	traditional Chinese (serif)
japan	Gothic	Japanese
japan-s	Mincho	Japanese (serif)
korea	Dotum	Korean
korea-s	Batang	Korean (serif)

insertImage (*rect*, *filename=None*, *pixmap=None*, *stream=None*, *mask=None*, *rotate=0*, *oc=0*, *keep_proportion=True*, *overlay=True*)

PDF only: Put an image inside the given rectangle. The image can be taken from a pixmap, a file or a memory area - of these parameters **exactly one** must be specified.

Changed in version 1.14.11 By default, the image keeps its aspect ratio.

Parameters

- **rect** (*rect_like*) – where to put the image. Must be finite and not empty.

¹ If your existing code already uses the installed base name as a font reference (as it was supported by PyMuPDF versions earlier than 1.14), this will continue to work.

³ Not all PDF readers display these fonts at all. Some others do, but use a wrong character spacing, etc.

² Not all PDF reader software (including internet browsers and office software) display all of these fonts. And if they do, the difference between the **serifed** and the **non-serifed** version may hardly be noticeable. But serifed and non-serifed versions lead to different installed base fonts, thus providing an option to be displayable with your specific PDF viewer.

(Changed in v1.17.6) No longer needs to have a non-empty intersection with the page's `Page.CropBox`⁵.

(Changed in version 1.14.13) The image is now always placed **centered** in the rectangle, i.e. the centers of image and rectangle are equal.

- **filename** (*str*) – name of an image file (all formats supported by MuPDF – see *Supported Input Image Formats*). If the same image is to be inserted multiple times, choose one of the other two options to avoid some overhead.
- **stream** (*bytes, bytearray, io.BytesIO*) – image in memory (all formats supported by MuPDF – see *Supported Input Image Formats*). This is the most efficient option.

Changed in version 1.14.13: *io.BytesIO* is now also supported.

- **pixmap** (*Pixmap*) – a pixmap containing the image.
- **mask** (*bytes, bytearray, io.BytesIO*) – (new in version v1.18.1) image in memory – to be used as image mask for the base image. When specified, the base image must also be provided as an in-memory image (*stream* parameter).
- **rotate** (*int*) – (new in version v1.14.11) rotate the image. Must be an integer multiple of 90 degrees. If you need a rotation by an arbitrary angle, consider converting the image to a PDF (`Document.convertToPDF()`) first and then use `Page.showPDFpage()` instead.
- **oc** (*int*) – (new in v1.18.3) (*xref*) make image visibility dependent on this OCG (optional content group). Please be aware, that this property is stored with the generated PDF image definition. If you insert the same image anywhere else, but **with a different 'oc' value**, a full additional image copy will be stored.
- **keep_proportion** (*bool*) – (new in version v1.14.11) maintain the aspect ratio of the image.

For a description of *overlay* see *Common Parameters*.

This example puts the same image on every page of a document:

```
>>> doc = fitz.open(...)
>>> rect = fitz.Rect(0, 0, 50, 50)      # put thumbnail in upper left corner
>>> img = open("some.jpg", "rb").read() # an image file
>>> for page in doc:
>>>     page.insertImage(rect, stream = img)
>>> doc.save(...)
```

Note:

1. If that same image had already been present in the PDF, then only a reference to it will be inserted. This of course considerably saves disk space and processing time. But to detect this fact, existing PDF images need to be compared with the new one. This is achieved by storing an MD5 code for each image in a table and only compare the new image's MD5 code against the table entries. Generating this MD5 table, however, is done when the first image is inserted - which therefore may have an extended response time.
2. You can use this method to provide a background or foreground image for the page, like a copyright, a watermark. Please remember, that watermarks require a transparent image ...

⁵ The previous algorithm caused images to be **shrunk** to this intersection. Now the image can be anywhere on `Page.MediaBox`, potentially being invisible or only partially visible if the cropbox (representing the visible page part) is smaller.

3. The image may be inserted uncompressed, e.g. if a *Pixmap* is used or if the image has an alpha channel. Therefore, consider using *deflate=True* when saving the file.
4. The image is stored in the PDF in its original quality. This may be much better than you ever need for your display. In this case consider decreasing the image size before inserting it – e.g. by using the *pixmap* option and then shrinking it or scaling it down (see *Pixmap* chapter). The PIL method *Image.thumbnail()* can also be used for that purpose. The file size savings can be very significant.
5. The most efficient way to display the same image on multiple pages is another method: *showPDFpage()*. Consult *Document.convertToPDF()* for how to obtain intermediary PDFs usable for that method. Demo script *fitz-logo.py* implements a fairly complete approach.

getText (*opt="text", clip=None, flags=None*)

Retrieves the content of a page in a variety of formats. This is a wrapper for *TextPage* methods by choosing the output option as follows:

- “text” – *TextPage.extractTEXT()*, default
- “blocks” – *TextPage.extractBLOCKS()*
- “words” – *TextPage.extractWORDS()*
- “html” – *TextPage.extractHTML()*
- “xhtml” – *TextPage.extractXHTML()*
- “xml” – *TextPage.extractXML()*
- “dict” – *TextPage.extractDICT()*
- “json” – *TextPage.extractJSON()*
- “rawdick” – *TextPage.extractRAWDICT()*
- “rawjson” – *TextPage.extractRAWJSON()*

Parameters

- **opt** (*str*) – A string indicating the requested format, one of the above. A mixture of upper and lower case is supported.

Changed in version 1.16.3 Values “words” and “blocks” are now also accepted.

- **clip** (*rect-like*) – (*new in v1.17.7*) restrict extracted text to this rectangle. If *None*, the full page is taken. Has **no effect** for options “html”, “xhtml” and “xml”.
- **flags** (*int*) – (*new in version 1.16.2*) indicator bits to control whether to include images or how text should be handled with respect to white spaces and ligatures. See *Preserve Text Flags* for available indicators and *Text Extraction Flags Defaults* for default settings.

Return type *str, list, dict*

Returns The page’s content as a string, a list or a dictionary. Refer to the corresponding *TextPage* method for details.

Note:

1. You can use this method as a **document conversion tool** from any supported document type (not only PDF!) to one of TEXT, HTML, XHTML or XML documents.

- The inclusion of text via the *clip* parameter is decided on a by-character level: **(changed in v1.18.2)** a character becomes part of the output, if its bbox is contained in *clip*. This **deviates** from the algorithm used in redaction annotations: a character will be removed if its bbox intersects with some redaction annotation.
-

getTextbox (*rect*)
(New in v1.17.7)

Retrieve the text contained in a rectangle.

Parameters *rect* (*rect-like*) – rect-like.

Returns

a string with interspersed linebreaks where necessary. This is the same as `page.getText("text", clip=rect, flags=0)` with one removed final line break. A typical use is checking the result of `Page.searchFor()`:

```
>>> rl = page.searchFor("currency:")
>>> page.getTextbox(rl[0])
'Currency: '
>>>
```

getTextPage (*clip=None, flags=3*)
(New in version 1.16.5)

Create a *TextPage* for the page. This method avoids using an intermediate *DisplayList*.

Parameters

- flags** (*in*) – indicator bits controlling the content available for subsequent extraction – see the parameter of `Page.getText()`.
- clip** (*rect-like*) – (new in v1.17.7) restrict extracted text to this area – to be used by text extraction methods.

Returns *TextPage*

getDrawings ()
(New in v1.18.0)

Return the draw commands of the page. These are instructions which draw lines, rectangles or curves, including properties like colors, transparency, line width and dashing, etc.

Returns a list of dictionaries. Each dictionary item contains one or more single draw commands which belong together: their lines are connected and they have the same properties (colors, dashing, etc.). This is called a “**path**” in the PDF specification, but the method works the same for **all document types**.

The path dictionary has been designed to be compatible with the methods and terminology of class *Shape*. There are the following keys:

Key	Value
closePath	Same as the parameter in <i>Shape</i> .
color	Same as the parameter in <i>Shape</i> .
dashes	Same as the parameter in <i>Shape</i> .
even_odd	Same as the parameter in <i>Shape</i> .
fill	Same as the parameter in <i>Shape</i> .
items	List of draw commands: lines, rectangle or curves.
lineCap	Number 3-tuple, use its max value on output with <i>Shape</i> .
lineJoin	Same as the parameter in <i>Shape</i> .
opacity	represents <i>fill_opacity</i> and <i>stroke_opacity</i> in <i>Shape</i> .
rect	Page area covered by this path. Information only.
width	Same as the parameter in <i>Shape</i> .

Each entry in `path["items"]` is one of the following:

- ("l", p1, p2) - a line from p1 to p2 (*Point* objects).
- ("c", p1, p2, p3, p4) - cubic Bézier curve from p1 to p4, p2 and p3 are the control points. All objects are of type *Point*.
- ("re", rect) - a *Rect*.

Using class *Shape*, you should be able to recreate the original drawings on a separate (PDF) page with high fidelity. A coding draft can be found in section “Extractings Drawings” of chapter *Collection of Recipes*.

The following limitations exist by design:

- The visual appearance of a page may have been designed in a very complex way. For example in PDF, layers (Optional Content Groups) can control the visibility of any item (drawings and other objects) depending on whatever condition: a watermark may be suppressed if the page is shown by a viewer, but is visible if printed on paper.
- Only drawings are extracted, other page content is ignored. The method therefore does not detect whether a drawing is covered, hidden or overlaid in the original document, e.g. by some text or an image.

Effects like these are ignored by the method – it will return all paths unconditionally.

getFontList (*full=False*)

PDF only: Return a list of fonts referenced by the page. Wrapper for *Document.getPageFontList()*.

getImageList (*full=False*)

PDF only: Return a list of images referenced by the page. Wrapper for *Document.getPageImageList()*.

getImageBbox (*item*)

PDF only: Return the boundary box of an image.

Changed in version 1.17.0:

- The method should deliver correct results now.
- The page’s `/Contents` are no longer modified by this method.

Parameters *item* (*list, str*) – an item of the list *Page.getImageList()* with *full=True* specified, or the **name** entry of such an item, which is `item[-3]` (or `item[7]` respectively).

Return type *Rect*

Returns the boundary box of the image. (*Changed in v1.16.7*) – If the page in fact does not display this image, an infinite rectangle is returned now. In previous versions, an exception was raised. (*Changed in v1.17.0*) – Only images referenced directly by the page are considered. This means that images occurring in embedded PDF pages are ignored and an exception is raised. (*Changed in v1.18.5*) – Removed the restriction introduced in v1.17.0. The base MuPDF library now more reliably computes that value.

Note:

- Be aware that `Page.getImageList()` may contain “dead” entries, i.e. images **not displayed** by this page (some PDFs contain a central list of all images, to save specification effort on the page level). In this case an infinite rectangle is returned.
-

getSVGimage (*matrix=fitz.Identity, text_as_path=True*)

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

Parameters

- **matrix** (*matrix_like*) – a matrix, default is *Identity*.
- **text_as_path** (*bool*) – (*new in v1.17.5*) – controls how text is represented. *True* outputs each character as a series of elementary draw commands, which leads to a more precise text display in browsers, but a **very much larger** output for text-oriented pages. Display quality for *False* relies on the presence of the referenced fonts on the current system. For missing fonts, the internet browser will fall back to some default – leading to unpleasant appearances. Choose *False* if you want to parse the text of the SVG.

Returns a UTF-8 encoded string that contains the image. Because SVG has XML syntax it can be saved in a text file with extension `.svg`.

getPixmap (*matrix=fitz.Identity, colorspace=fitz.csRGB, clip=None, alpha=False, annots=True*)

Create a pixmap from the page. This is probably the most often used method to create a *Pixmap*.

Parameters

- **matrix** (*matrix_like*) – default is *Identity*.
- **colorspace** (*str* or *Colorspace*) – Defines the required colorspace, one of “GRAY”, “RGB” or “CMYK” (case insensitive). Or specify a *Colorspace*, i.e. one of the predefined ones: *csGRAY*, *csRGB* or *csCMYK*.
- **clip** (*irect_like*) – restrict rendering to this area.
- **alpha** (*bool*) – whether to add an alpha channel. Always accept the default *False* if you do not really need transparency. This will save a lot of memory (25% in case of RGB ... and pixmaps are typically **large!**), and also processing time. Also note an **important difference** in how the image will be rendered: with *True* the pixmap’s samples area will be pre-cleared with `0x00`. This results in **transparent** areas where the page is empty. With *False* the pixmap’s samples will be pre-cleared with `0xff`. This results in **white** where the page has nothing to show.

Changed in version 1.14.17 The default alpha value is now *False*.

- Generated with *alpha=True*



– Generated with `alpha=False`



- **annots** (*bool*) – (new in version 1.16.0) whether to also render annotations or to suppress them. You can create pixmaps for annotations separately.

Return type *Pixmap*

Returns Pixmap of the page. For fine-controlling the generated image, the by far most important parameter is **matrix**. E.g. you can increase or decrease the image resolution by using **Matrix(xzoom, yzoom)**. If `zoom > 1`, you will get a higher resolution: `zoom=2` will double the number of pixels in that direction and thus generate a 2 times larger image. Non-positive values will flip horizontally, resp. vertically. Similarly, matrices also let you rotate or shear, and you can combine effects via e.g. matrix multiplication. See the *Matrix* section to learn more.

annot_names ()

(New in version 1.16.10)

PDF only: return a list of the names of annotations, widgets and links. Technically, these are the */NM* values of every PDF object found in the page's */Annots* array.

Return type list

annot_xrefs ()

(New in version 1.17.1)

PDF only: return a list of the `:data'xref'` numbers of annotations, widgets and links – technically of all entries found in the page's */Annots* array.

Return type list

Returns a list of items (*xref, type*) where *type* is the annotation type. Use the *type* to tell apart links, fields and annotations, see *Annotation Types*.

load_annot (*ident*)

(Deprecated since v1.17.1).

loadAnnot (*ident*)*(New in version 1.17.1)*

PDF only: return the annotation identified by *ident*. This may be its unique name (PDF */NM* key), or its *xref*.

Parameters *ident* (*str*, *int*) – the annotation name or xref.

Return type *Annot*

Returns the annotation or *None*.

Note: Methods `Page.annot_names()`, `Page.annots_xrefs()` provide lists of names or xrefs, respectively, from where an item may be picked and loaded via this method.

loadLinks ()

Return the first link on a page. Synonym of property *firstLink*.

Return type *Link*

Returns first link on the page (or *None*).

setRotation (*rotate*)

PDF only: Sets the rotation of the page.

Parameters *rotate* (*int*) – An integer specifying the required rotation in degrees. Must be an integer multiple of 90. Values will be converted to one of 0, 90, 180, 270.

showPDFpage (*rect*, *docsrc*, *pno=0*, *keep_proportion=True*, *overlay=True*, *oc=0*, *rotate=0*, *clip=None*)

PDF only: Display a page of another PDF as a **vector image** (otherwise similar to `Page.insertImage()`). This is a multi-purpose method. For example, you can use it to

- create “n-up” versions of existing PDF files, combining several input pages into **one output page** (see example `4-up.py`),
- create “posterized” PDF files, i.e. every input page is split up in parts which each create a separate output page (see `posterize.py`),
- include PDF-based vector images like company logos, watermarks, etc., see `svg-logo.py`, which puts an SVG-based logo on each page (requires additional packages to deal with SVG-to-PDF conversions).

Changed in version 1.14.11 Parameter *reuse_xref* has been deprecated.

Parameters

- **rect** (*rect_like*) – where to place the image on current page. Must be finite and its intersection with the page must not be empty.

Changed in version 1.14.11 Position the source rectangle centered in this rectangle.

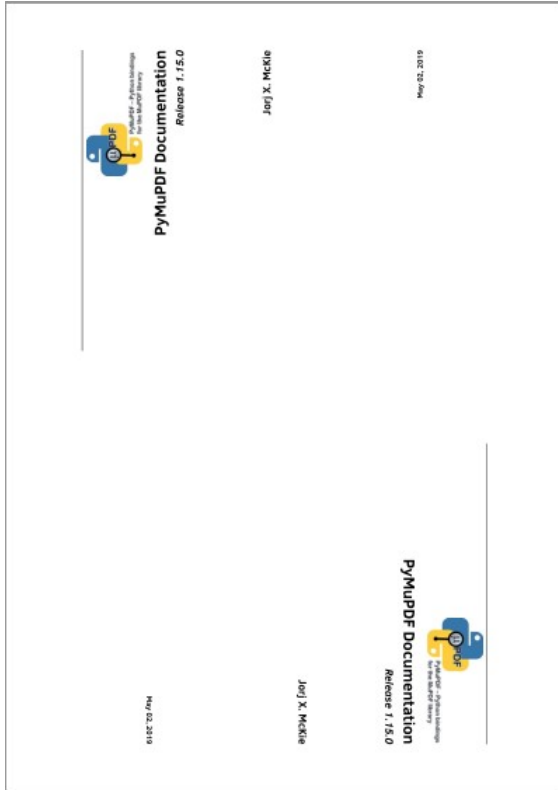
- **docsrc** (*Document*) – source PDF document containing the page. Must be a different document object, but may be the same file.
- **pno** (*int*) – page number (0-based, in $-inf < pno < docsrc.pageCount$) to be shown.
- **keep_proportion** (*bool*) – whether to maintain the width-height-ratio (default). If false, all 4 corners are always positioned on the border of the target rectangle – whatever the rotation value. In general, this will deliver distorted and /or non-rectangular images.

- **overlay** (*bool*) – put image in foreground (default) or background.
- **oc** (*int*) – (*new in v1.18.3*) (*xref*) make visibility dependent on this OCG (optional content group).
- **rotate** (*float*) – (*new in version 1.14.10*) show the source rectangle rotated by some angle. *Changed in version 1.14.11*: Any angle is now supported.
- **clip** (*rect_like*) – choose which part of the source page to show. Default is the full page, else must be finite and its intersection with the source page must not be empty.

Note: In contrast to method `Document.insertPDF()`, this method does not copy annotations or links, so they are not shown. But all its **other resources (text, images, fonts, etc.)** will be imported into the current PDF. They will therefore appear in text extractions and in `getFontList()` and `getImageList()` lists – even if they are not contained in the visible area given by `clip`.

Example: Show the same source page, rotated by 90 and by -90 degrees:

```
>>> doc = fitz.open() # new empty PDF
>>> page=doc.newPage() # new page in A4 format
>>>
>>> # upper half page
>>> r1 = fitz.Rect(0, 0, page.rect.width, page.rect.height/2)
>>>
>>> # lower half page
>>> r2 = r1 + (0, page.rect.height/2, 0, page.rect.height/2)
>>>
>>> src = fitz.open("PyMuPDF.pdf") # show page 0 of this
>>>
>>> page.showPDFpage(r1, src, 0, rotate=90)
>>> page.showPDFpage(r2, src, 0, rotate=-90)
>>> doc.save("show.pdf")
```

**newShape ()**

PDF only: Create a new *Shape* object for the page.

Return type *Shape*

Returns a new *Shape* to use for compound drawings. See description there.

searchFor (*needle*, *clip=clip*, *quads=False*, *flags=TEXT_DEHYPHENATE*)
(Changed in v1.18.2)

Search for *needle* on a page. Wrapper for *TextPage.search()*.

Parameters

- **needle** (*str*) – Text to search for. Upper / lower case is ignored. May contain spaces.
- **clip** (*rect_like*) – (New in v1.18.2) only search within this area.
- **quads** (*bool*) – Return object type *Quad* instead of *Rect*.
- **flags** (*int*) – Control the data extracted by the underlying *TextPage*. By default ligatures are expanded, white space is replaced with spaces and hyphenation is detected.

Return type list

Returns

A list of *Rect* or *Quad* objects, each of which – **normally!** – surrounds one occurrence of *needle*. **However:** if parts of *needle* occur on more than one line, then a separate item is generated for each these parts. So, if `needle = "search string"`, two rectangles may be generated.

Changes in v1.18.2:

- There no longer is a limit on the list length (removal of the `hit_max` parameter).
- If a word is **hyphenated** at a line break, it will still be found. E.g. the word “method” will be found even if hyphenated as “meth-od” by a line break, and two rectangles will be returned: one surrounding “meth” (without the hyphen) and another one surrounding “od”.

Note: The method supports multi-line text marker annotations: you can use the full returned list as **one single** parameter for creating the annotation.

Caution:

- There is a tricky aspect: the search logic regards **contiguous multiple occurrences** of *needle* as one: assuming *needle* is “abc”, and the page contains “abc” and “abcabc”, then only **two** rectangles will be returned, one for “abc”, and a second one for “abcabc”.
- You can always use `Page.getTextbox()` to check what text actually is being surrounded by each rectangle.

setMediaBox (*r*)

PDF only: (*New in v1.16.13*) Change the physical page dimension by setting *MediaBox* in the page’s object definition.

Parameters *r* (*rect-like*) – the new *MediaBox* value.

Note: This method also sets the page’s *CropBox* to the same value – to prevent mismatches caused by values further up in the parent hierarchy.

Caution: For existing pages this may have unexpected effects, if painting commands depend on a certain setting, and may lead to an empty or distorted appearance.

setCropBox (*r*)

PDF only: change the visible part of the page.

Parameters *r* (*rect_like*) – the new visible area of the page. Note that this **must** be specified in **unrotated coordinates**.

After execution if the page is not rotated, `Page.rect` will equal this rectangle, but shifted to the top-left position (0, 0) if necessary. Example session:

```
>>> page = doc.newPage()
>>> page.rect
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> page.CropBox                # CropBox and MediaBox still equal
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> # now set CropBox to a part of the page
>>> page.setCropBox(fitz.Rect(100, 100, 400, 400))
>>> # this will also change the "rect" property:
>>> page.rect
fitz.Rect(0.0, 0.0, 300.0, 300.0)
```

(continues on next page)

(continued from previous page)

```

>>>
>>> # but MediaBox remains unaffected
>>> page.MediaBox
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> # revert everything we did
>>> page.setCropBox(page.MediaBox)
>>> page.rect
fitz.Rect(0.0, 0.0, 595.0, 842.0)

```

rotation

Contains the rotation of the page in degrees (always 0 for non-PDF types).

Type *int*

CropBoxPosition

Contains the top-left point of the page's */CropBox* for a PDF, otherwise *Point(0, 0)*.

Type *Point*

CropBox

The page's */CropBox* for a PDF. Always the **unrotated** page rectangle is returned. For a non-PDF this will always equal the page rectangle.

Type *Rect*

MediaBoxSize

Contains the width and height of the page's *Page.MediaBox* for a PDF, otherwise the bottom-right coordinates of *Page.rect*.

Type *Point*

MediaBox

The page's *MediaBox* for a PDF, otherwise *Page.rect*.

Type *Rect*

Note: For most PDF documents and for **all other document types**, *page.rect == page.CropBox == page.MediaBox* is true. However, for some PDFs the visible page is a true subset of *MediaBox*. Also, if the page is rotated, its *Page.rect* may not equal *Page.CropBox*. In these cases the above attributes help to correctly locate page elements.

transformationMatrix

This matrix translates coordinates from the PDF space to the MuPDF space. For example, in PDF */Rect [x0 y0 x1 y1]* the pair (x0, y0) specifies the **bottom-left** point of the rectangle – in contrast to MuPDF's system, where (x0, y0) specify top-left. Multiplying the PDF coordinates with this matrix will deliver the (Py-) MuPDF rectangle version. Obviously, the inverse matrix will again yield the PDF rectangle.

Type *Matrix*

rotationMatrix**derotationMatrix**

These matrices may be used for dealing with rotated PDF pages. When adding / inserting anything to a PDF page with PyMuPDF, the coordinates of the **unrotated** page are always used. These matrices help translating between the two states. Example: if a page is rotated by 90 degrees – what would then be the coordinates of the top-left *Point(0, 0)* of an A4 page?

```

>>> page.setRotation(90) # rotate an ISO A4 page
>>> page.rect
Rect(0.0, 0.0, 842.0, 595.0)
>>> p = fitz.Point(0, 0) # where did top-left point land?
>>> p * page.rotationMatrix
Point(842.0, 0.0)
>>>

```

Type *Matrix*

firstLink

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

Type *Link*

firstAnnot

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

Type *Annot*

firstWidget

Contains the first *Widget* of a page (or *None*).

Type *Widget*

number

The page number.

Type *int*

parent

The owning document object.

Type *Document*

rect

Contains the rectangle of the page. Same as result of *Page.bound()*.

Type *Rect*

xref

The page's PDF *xref*. Zero if not a PDF.

Type *Rect*

6.12.2 Description of *getLinks()* Entries

Each entry of the *getLinks()* list is a dictionary with the following keys:

- *kind*: (required) an integer indicating the kind of link. This is one of *LINK_NONE*, *LINK_GOTO*, *LINK_GOTOR*, *LINK_LAUNCH*, or *LINK_URI*. For values and meaning of these names refer to *Link Destination Kinds*.
- *from*: (required) a *Rect* describing the “hot spot” location on the page’s visible representation (where the cursor changes to a hand image, usually).
- *page*: a 0-based integer indicating the destination page. Required for *LINK_GOTO* and *LINK_GOTOR*, else ignored.

- *to*: either a *fitz.Point*, specifying the destination location on the provided page, default is *fitz.Point(0, 0)*, or a symbolic (indirect) name. If an indirect name is specified, *page = -1* is required and the name must be defined in the PDF in order for this to work. Required for *LINK_GOTO* and *LINK_GOTOR*, else ignored.
- *file*: a string specifying the destination file. Required for *LINK_GOTOR* and *LINK_LAUNCH*, else ignored.
- *uri*: a string specifying the destination internet resource. Required for *LINK_URI*, else ignored.
- *xref*: an integer specifying the PDF *xref* of the link object. Do not change this entry in any way. Required for link deletion and update, otherwise ignored. For non-PDF documents, this entry contains *-1*. It is also *-1* for **all** entries in the *getLinks()* list, if **any** of the links is not supported by MuPDF - see the note below.

6.12.3 Notes on Supporting Links

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

6.12.3.1 Reading (pertains to method *getLinks()* and the *firstLink* property chain)

If MuPDF detects a link to another file, it will supply either a *LINK_GOTOR* or a *LINK_LAUNCH* link kind. In case of *LINK_GOTOR* destination details may either be given as page number (eventually including position information), or as an indirect destination.

If an indirect destination is given, then this is indicated by *page = -1*, and *link.dest.dest* will contain this name. The dictionaries in the *getLinks()* list will contain this information as the *to* value.

Internal links are always of kind *LINK_GOTO*. If an internal link specifies an indirect destination, it **will always be resolved** and the resulting direct destination will be returned. Names are **never returned for internal links**, and undefined destinations will cause the link to be ignored.

6.12.3.2 Writing

PyMuPDF writes (updates, inserts) links by constructing and writing the appropriate PDF object **source**. This makes it possible to specify indirect destinations for *LINK_GOTOR* and *LINK_GOTO* link kinds (pre *PDF 1.2* file formats are **not supported**).

Warning: If a *LINK_GOTO* indirect destination specifies an undefined name, this link can later on not be found / read again with MuPDF / PyMuPDF. Other readers however **will** detect it, but flag it as erroneous.

Indirect *LINK_GOTOR* destinations can in general of course not be checked for validity and are therefore **always accepted**.

6.12.4 Homologous Methods of Document and Page

This is an overview of homologous methods on the *Document* and on the *Page* level.

Document Level	Page Level
<i>Document.getPageFontlist(pno)</i>	<i>Page.getFontList()</i>
<i>Document.getPageImageList(pno)</i>	<i>Page.getImageList()</i>
<i>Document.getPagePixmap(pno, ...)</i>	<i>Page.getPixmap()</i>
<i>Document.getPageText(pno, ...)</i>	<i>Page.getText()</i>
<i>Document.searchPageFor(pno, ...)</i>	<i>Page.searchFor()</i>

The page number “pno” is a 0-based integer $-inf < pno < pageCount$.

Note: Most document methods (left column) exist for convenience reasons, and are just wrappers for: *Document[pno].<page method>*. So they **load and discard the page** on each execution.

However, the first two methods work differently. They only need a page’s object definition statement - the page itself will **not** be loaded. So e.g. *Page.getFontList()* is a wrapper the other way round and defined as follows: *page.getFontList == page.parent.getPageFontList(page.number)*.

6.13 Pixmap

Pixmaps (“pixel maps”) are objects at the heart of MuPDF’s rendering capabilities. They represent plane rectangular sets of pixels. Each pixel is described by a number of bytes (“components”) defining its color, plus an optional alpha byte defining its transparency.

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

1. from a document page (method *Page.getPixmap()*)
2. empty, based on *Colorspace* and *IRect* information
3. from a file
4. from an in-memory image
5. from a memory area of plain pixels
6. from an image inside a PDF document
7. as a copy of another pixmap

Note: A number of image formats is supported as input for points 3. and 4. above. See section *Supported Input Image Formats*.

Have a look at the *Collection of Recipes* section to see some pixmap usage “at work”.

Method / Attribute	Short Description
<i>Pixmap.clearWith()</i>	clear parts of a pixmap
<i>Pixmap.copyPixmap()</i>	copy parts of another pixmap
<i>Pixmap.gammaWith()</i>	apply a gamma factor to the pixmap
<i>Pixmap.getImageData()</i>	return a memory area in a variety of formats
<i>Pixmap.getPNGData()</i>	return a PNG as a memory area
<i>Pixmap.invertIRect()</i>	invert the pixels of a given area
<i>Pixmap.pillowWrite()</i>	save as image using pillow (experimental)
<i>Pixmap.pillowData()</i>	write image stream using pillow (experimental)
<i>Pixmap.pixel()</i>	return the value of a pixel
<i>Pixmap.setAlpha()</i>	set alpha values
<i>Pixmap.setPixel()</i>	set the color of a pixel
<i>Pixmap.setRect()</i>	set the color of a rectangle
<i>Pixmap.setResolution()</i>	set the image resolution
<i>Pixmap.setOrigin()</i>	set pixmap x,y values

Continued on next page

Table 4 – continued from previous page

Method / Attribute	Short Description
<code>Pixmap.shrink()</code>	reduce size keeping proportions
<code>Pixmap.tintWith()</code>	tint a pixmap with a color
<code>Pixmap.writeImage()</code>	save a pixmap in a variety of formats
<code>Pixmap.writePNG()</code>	save a pixmap as a PNG file
<code>Pixmap.alpha</code>	transparency indicator
<code>Pixmap.colorspace</code>	pixmap's <i>Colorspace</i>
<code>Pixmap.height</code>	pixmap height
<code>Pixmap.interpolate</code>	interpolation method indicator
<code>Pixmap.irect</code>	<i>IRect</i> of the pixmap
<code>Pixmap.n</code>	bytes per pixel
<code>Pixmap.samples</code>	pixel area
<code>Pixmap.size</code>	pixmap's total length
<code>Pixmap.stride</code>	size of one image row
<code>Pixmap.width</code>	pixmap width
<code>Pixmap.x</code>	X-coordinate of top-left corner
<code>Pixmap.xres</code>	resolution in X-direction
<code>Pixmap.y</code>	Y-coordinate of top-left corner
<code>Pixmap.yres</code>	resolution in Y-direction

Class API

class Pixmap

`__init__(self, colorspace, irect, alpha)`

New empty pixmap: Create an empty pixmap of size and origin given by the rectangle. So, `irect.top_left` designates the top left corner of the pixmap, and its width and height are `irect.width` resp. `irect.height`. Note that the image area is **not initialized** and will contain crap data – use eg. `clearWith()` or `setRect()` to be sure.

Parameters

- **colorspace** (*Colorspace*) – colorspace.
- **irect** (*irect_like*) – Tte pixmap's position and dimension.
- **alpha** (*bool*) – Specifies whether transparency bytes should be included. Default is *False*.

`__init__(self, colorspace, source)`

Copy and set colorspace: Copy *source* pixmap converting colorspace. Any colorspace combination is possible, but source colorspace must not be *None*.

Parameters

- **colorspace** (*Colorspace*) – desired **target** colorspace. This **may also be None**. In this case, a “masking” pixmap is created: its `Pixmap.samples` will consist of the source's alpha bytes only.
- **source** (*Pixmap*) – the source pixmap.

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

Copy and scale: Copy *source* pixmap, scaling new width and height values – the image will appear stretched or shrunk accordingly. Supports partial copying. The source colorspace may be *None*.

Parameters

- **source** (*Pixmap*) – the source pixmap.

- **width** (*float*) – desired target width.
- **height** (*float*) – desired target height.
- **clip** (*irect_like*) – restrict the resulting pixmap to this region of the **scaled** pixmap.

Note: If width or height are not *de facto* integers (i.e. `float(int(value)) != value`), then the resulting pixmap will have an alpha channel.

`__init__` (*self, source, alpha=1*)

Copy and add or drop alpha: Copy *source* and add or drop its alpha channel. Identical copy if *alpha* equals *source.alpha*. If an alpha channel is added, its values will be set to 255.

Parameters

- **source** (*Pixmap*) – source pixmap.
- **alpha** (*bool*) – whether the target will have an alpha channel, default and mandatory if source colorspace is *None*.

Note: A typical use includes separation of color and transparency bytes in separate pixmaps. Some applications require this like e.g. `wx.Bitmap.FromBufferAndAlpha()` of `wxPython`:

```
>>> # 'pix' is an RGBA pixmap
>>> pixcolors = fitz.Pixmap(pix, 0) # extract the RGB part (drop alpha)
>>> pixalpha = fitz.Pixmap(None, pix) # extract the alpha part
>>> bm = wx.Bitmap.FromBufferAndAlpha(pix.widht, pix.height, pixcolors,
↳samples, pixalpha.samples)
```

`__init__` (*self, filename*)

From a file: Create a pixmap from *filename*. All properties are inferred from the input. The origin of the resulting pixmap is (0, 0).

Parameters filename (*str*) – Path of the image file.

`__init__` (*self, stream*)

From memory: Create a pixmap from a memory area. All properties are inferred from the input. The origin of the resulting pixmap is (0, 0).

Parameters stream (*bytes, bytearray, BytesIO*) – Data containing a complete, valid image. Could have been created by e.g. `stream = bytearray(open('image.file', 'rb').read())`. Type *bytes* is supported in **Python 3 only**, because `bytes == str` in Python 2 and the method will interpret the stream as a filename.

Changed in version 1.14.13: io.BytesIO is now also supported.

`__init__` (*self, colorspace, width, height, samples, alpha*)

From plain pixels: Create a pixmap from *samples*. Each pixel must be represented by a number of bytes as controlled by the *colorspace* and *alpha* parameters. The origin of the resulting pixmap is (0, 0). This method is useful when raw image data are provided by some other program – see *Collection of Recipes*.

Parameters

- **colorspace** (*Colorspace*) – Colorspace of image.
- **width** (*int*) – image width
- **height** (*int*) – image height

- **samples** (*bytes, bytearray, BytesIO*) – an area containing all pixels of the image. Must include alpha values if specified.

Changed in version 1.14.13: (1) *io.BytesIO* can now also be used. (2) Data are now **copied** to the pixmap, so may safely be deleted or become unavailable.

- **alpha** (*bool*) – whether a transparency channel is included.
-

Note:

1. The following equation **must be true**: $(\text{colorspace.n} + \text{alpha}) * \text{width} * \text{height} == \text{len}(\text{samples})$.
 2. Starting with version 1.14.13, the samples data are **copied** to the pixmap.
-

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

From a PDF image: Create a pixmap from an image **contained in PDF** *doc* identified by its *xref*. All pixmap properties are set by the image. Have a look at [extract-img1.py](#) and [extract-img2.py](#) to see how this can be used to recover all of a PDF's images.

Parameters

- **doc** (*Document*) – an opened **PDF** document.
- **xref** (*int*) – the *xref* of an image object. For example, you can make a list of images used on a particular page with `Document.getPageImageList()`, which also shows the *xref* numbers of each image.

`clearWith` (*[value[, irect]]*)

Initialize the samples area.

Parameters

- **value** (*int*) – if specified, values from 0 to 255 are valid. Each color byte of each pixel will be set to this value, while alpha will be set to 255 (non-transparent) if present. If omitted, then all bytes (including any alpha) are cleared to `0x00`.
- **irect** (*irect_like*) – the area to be cleared. Omit to clear the whole pixmap. Can only be specified, if *value* is also specified.

`tintWith` (*red, green, blue*)

Colorize (tint) a pixmap with a color provided as an integer triple (red, green, blue). Only colorspaces `CS_GRAY` and `CS_RGB` are supported, others are ignored with a warning.

If the colorspace is `CS_GRAY`, $(\text{red} + \text{green} + \text{blue})/3$ will be taken as the tint 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. Pixmaps with colorspace `None` are ignored with a warning.

Parameters **gamma** (*float*) – *gamma* = 1.0 does nothing, *gamma* < 1.0 lightens, *gamma* > 1.0 darkens the image.

`shrink` (*n*)

Shrink the pixmap by dividing both, its width and height by 2^n .

Parameters *n* (*int*) – determines the new pixmap (samples) size. For example, a value of 2 divides width and height by 4 and thus results in a size of one 16th of the original. Values less than 1 are ignored with a warning.

Note: Use this methods to reduce a pixmap’s size retaining its proportion. The pixmap is changed “in place”. If you want to keep original and also have more granular choices, use the resp. copy constructor above.

pixel (*x*, *y*)

New in version: 1.14.5: Return the value of the pixel at location (*x*, *y*) (column, line).

Parameters

- **x** (*int*) – the column number of the pixel. Must be in *range*(*pix.width*).
- **y** (*int*) – the line number of the pixel, Must be in *range*(*pix.height*).

Return type list

Returns a list of color values and, potentially the alpha value. Its length and content depend on the pixmap’s colorspace and the presence of an alpha. For RGBA pixmaps the result would e.g. be [*r*, *g*, *b*, *a*]. All items are integers in *range*(256).

setPixel (*x*, *y*, *color*)

New in version 1.14.7: Set the color of the pixel at location (*x*, *y*) (column, line).

Parameters

- **x** (*int*) – the column number of the pixel. Must be in *range*(*pix.width*).
- **y** (*int*) – the line number of the pixel. Must be in *range*(*pix.height*).
- **color** (*sequence*) – the desired color given as a sequence of integers in *range*(256). The length of the sequence must equal *Pixmap.n*, which includes any alpha byte.

setRect (*irect*, *color*)

New in version 1.14.8: Set the pixels of a rectangle to a color.

Parameters

- **irect** (*irect_like*) – the rectangle to be filled with the color. The actual area is the intersection of this parameter and *Pixmap.irect*. For an empty intersection (or an invalid parameter), no change will happen.
- **color** (*sequence*) – the desired color given as a sequence of integers in *range*(256). The length of the sequence must equal *Pixmap.n*, which includes any alpha byte.

Return type bool

Returns *False* if the rectangle was invalid or had an empty intersection with *Pixmap.irect*, else *True*.

Note:

1. This method is equivalent to *Pixmap.setPixel()* executed for each pixel in the rectangle, but is obviously **very much faster** if many pixels are involved.

- This method can be used similar to `Pixmap.clearWith()` to initialize a pixmap with a certain color like this: `pix.setRect(pix.irect, (255, 255, 0))` (RGB example, colors the complete pixmap with yellow).
-

setOrigin (*x*, *y*)

(New in v1.17.7) Set the x and y values.

Parameters

- **x** (*int*) – x coordinate
- **y** (*int*) – y coordinate

setResolution (*xres*, *yres*)

(New in v1.16.17) Set the resolution (dpi) in x and y direction.

(Changed in v1.18.0) When saving as a PNG image, these values will be stored now.

Parameters

- **xres** (*int*) – resolution in x direction.
- **yres** (*int*) – resolution in y direction.

setAlpha ([*alphavalues*])

Change the alpha values. The pixmap must have an alpha channel.

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

Changed in version 1.14.13: *io.BytesIO* is now also supported.

invertIRect ([*irect*])

Invert the color of all pixels in *IRect irect*. Will have no effect if colorspace is *None*.

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

copyPixmap (*source*, *irect*)

Copy the *irect* part of the *source* pixmap into the corresponding area of this one. The two pixmaps may have different dimensions and can each have *CS_GRAY* or *CS_RGB* colorspaces, but they currently **must** have the same alpha property². The copy mechanism automatically adjusts discrepancies between source and target like so:

If copying from *CS_GRAY* to *CS_RGB*, the source gray-shade value will be put into each of the three rgb component bytes. If the other way round, $(r + g + b) / 3$ will be taken as the gray-shade value of the target.

Between *irect* and the target pixmap's rectangle, an "intersection" is calculated at first. This takes into account the rectangle coordinates and the current attribute values *source.x* and *source.y* (which you are free to modify for this purpose). Then the corresponding data of this intersection are copied. If the intersection is empty, nothing will happen.

Parameters

- **source** (*Pixmap*) – source pixmap.
- **irect** (*irect_like*) – The area to be copied.

writeImage (*filename*, *output=None*)

Save pixmap as an image file. Depending on the output chosen, only some or all colorspaces are supported

² To also set the alpha property, add an additional step to this method by dropping or adding an alpha channel to the result.

and different file extensions can be chosen. Please see the table below. Since MuPDF v1.10a the *savealpha* option is no longer supported and will be silently ignored.

Parameters

- **filename** (*str*) – The filename to save to. The filename’s extension determines the image format, if not overridden by the output parameter.
- **output** (*str*) – The requested image format. The default is the filename’s extension. If not recognized, *png* is assumed. For other possible values see *Supported Output Image Formats*.

writePNG (*filename*)

Equal to `pix.writeImage(filename, "png")`.

getImageData (*output="png"*)

New in version 1.14.5: Return the pixmap as a *bytes* memory object of the specified format – similar to `writeImage()`.

Parameters **output** (*str*) – The requested image format. The default is “png” for which this function equals `getPNGData()`. For other possible values see *Supported Output Image Formats*.

Return type bytes

getPNGdata ()

getPNGData ()

Equal to `pix.getImageData("png")`.

Return type bytes

pillowWrite (**args, **kwargs*)

(New in v1.17.3)

Write the pixmap as an image file using Pillow. Use this method for image formats or extended image features not supported by MuPDF. Examples are

- Formats JPEG, JPX, J2K, WebP, etc.
- Storing EXIF information.
- If you do not provide dpi information, the values *xres*, *yres* stored with the pixmap are automatically used.

A simple example: `pix.pillowWrite("some.jpg", optimize=True, dpi=(150, 150))`. For details on other parameters see the Pillow documentation.

Note: *(Changed in v1.18.0)* `Pixmap.writeImage()` and `Pixmap.writePNG()` now also set resolution / dpi from *xres* / *yres* automatically, when saving a PNG image.

pillowData (**args, **kwargs*)

(New in v1.17.3)

Return an image as a bytes object in the specified format using Pillow. For example `stream = pix.pillowData(format="JPEG", optimize=True)`. Also see above. For details on other parameters see the Pillow documentation.

alpha

Indicates whether the pixmap contains transparency information.

Type bool

colorspace

The colorspace of the pixmap. This value may be *None* if the image is to be treated as a so-called *image mask* or *stencil mask* (currently happens for extracted PDF document images only).

Type *Colorspace*

stride

Contains the length of one row of image data in *Pixmap.samples*. This is primarily used for calculation purposes. The following expressions are true:

- $len(samples) == height * stride$
- $width * n == stride$.

Type int

irect

Contains the *IRect* of the pixmap.

Type *IRect*

samples

The color and (if *Pixmap.alpha* is true) transparency values for all pixels. It is an area of $width * height * n$ bytes. Each n bytes define one pixel. Each successive n bytes yield another pixel in scanline order. Subsequent scanlines follow each other with no padding. E.g. for an RGBA colorspace this means, *samples* is a sequence of bytes like \dots, R, G, B, A, \dots , and the four byte values R, G, B, A define one pixel.

This area can be passed to other graphics libraries like PIL (Python Imaging Library) to do additional processing like saving the pixmap in other image formats.

Note:

- The underlying data is a typically **large** memory area from which a *bytes* copy is made for this attribute: for example an RGB-rendered letter page has a samples size of almost 1.4 MB. So consider assigning a new variable if you repeatedly use it.
 - Any changes to the underlying data are available only after again accessing this attribute.
-

Type bytes

size

Contains $len(pixmap)$. This will generally equal $len(pix.samples)$ plus some platform-specific value for defining other attributes of the object.

Type int

width**w**

Width of the region in pixels.

Type int

height**h**

Height of the region in pixels.

Type int

x	X-coordinate of top-left corner Type int
y	Y-coordinate of top-left corner Type int
n	Number of components per pixel. This number depends on colorspace and alpha. If colorspace is not <i>None</i> (stencil masks), then <i>Pixmap.n - Pixmap.alpha == pixmap.colorsplace.n</i> is true. If colorspace is <i>None</i> , then <i>n == alpha == 1</i> . Type int
xres	Horizontal resolution in dpi (dots per inch). Please also see <i>resolution</i> . Type int
yres	Vertical resolution in dpi. Please also see <i>resolution</i> . Type int
interpolate	An information-only boolean flag set to <i>True</i> if the image will be drawn using “linear interpolation”. If <i>False</i> “nearest neighbour sampling” will be used. Type bool

6.13.1 Supported Input Image Formats

The following file types are supported as **input** to construct pixmaps: **BMP, JPEG, GIF, TIFF, JXR, JPX, PNG, PAM** and all of the **Portable Anymap** family (**PBM, PGM, PNM, PPM**). This support is two-fold:

1. Directly create a pixmap with *Pixmap(filename)* or *Pixmap(bytarray)*. The pixmap will then have properties as determined by the image.
2. Open such files with *fitz.open(...)*. The result will then appear as a document containing one single page. Creating a pixmap of this page offers all the options available in this context: apply a matrix, choose colorspace and alpha, confine the pixmap to a clip area, etc.

SVG images are only supported via method 2 above, not directly as pixmaps. But remember: the result of this is a **raster image** as is always the case with pixmaps¹.

6.13.2 Supported Output Image Formats

A number of image **output** formats are supported. You have the option to either write an image directly to a file (*Pixmap.writeImage()*), or to generate a bytes object (*Pixmap.getImageData()*). Both methods accept a 3-letter string identifying the desired format (**Format** column below). Please note that not all combinations of pixmap colorspace, transparency support (alpha) and image format are possible.

¹ If you need a **vector image** from the SVG, you must first convert it to a PDF. Try *Document.convertToPDF()*. If this is not good enough, look for other SVG-to-PDF conversion tools like the Python packages *svglib*, *CairoSVG*, *Uniconvertor* or the Java solution *Apache Batik*. Have a look at our Wiki for more examples.

Format	Colorspaces	alpha	Extensions	Description
pam	gray, rgb, cmyk	yes	.pam	Portable Arbitrary Map
pbm	gray, rgb	no	.pbm	Portable Bitmap
pgm	gray, rgb	no	.pgm	Portable Graymap
png	gray, rgb	yes	.png	Portable Network Graphics
pnm	gray, rgb	no	.pnm	Portable Anymap
ppm	gray, rgb	no	.ppm	Portable Pixmap
ps	gray, rgb, cmyk	no	.ps	Adobe PostScript Image
psd	gray, rgb, cmyk	yes	.psd	Adobe Photoshop Document

Note:

- Not all image file types are supported (or at least common) on all OS platforms. E.g. PAM and the Portable Anymap formats are rare or even unknown on Windows.
- Especially pertaining to CMYK colorspaces, you can always convert a CMYK pixmap to an RGB pixmap with `rgb_pix = fitz.Pixmap(fitz.csRGB, cmyk_pix)` and then save that in the desired format.
- As can be seen, MuPDF's image support range is different for input and output. Among those supported both ways, PNG is probably the most popular. We recommend using Pillow whenever you face a support gap.
- We also recommend using "ppm" formats as input to tkinter's `PhotoImage` method like this: `tkimg = tkinter.PhotoImage(data=pix.getImageData("ppm"))` (also see the tutorial). This is **very** fast (**60 times** faster than PNG) and will work under Python 2 or 3.

6.14 Point

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

Attribute / Method	Description
<code>Point.distance_to()</code>	calculate distance to point or rect
<code>Point.norm()</code>	the Euclidean norm
<code>Point.transform()</code>	transform point with a matrix
<code>Point.abs_unit</code>	same as unit, but positive coordinates
<code>Point.unit</code>	point coordinates divided by <code>abs(point)</code>
<code>Point.x</code>	the X-coordinate
<code>Point.y</code>	the Y-coordinate

Class API**class Point**

```
__init__(self)
```

```
__init__(self, x, y)
```

```
__init__(self, point)
```

```
__init__(self, sequence)
```

Overloaded constructors.

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

With another point specified, a **new copy** will be created, “sequence” is a Python sequence of 2 numbers (see *Using Python Sequences as Arguments in PyMuPDF*).

Parameters

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

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

Calculate the distance to *x*, which may be *point_like* or *rect_like*. The distance is given in units of either pixels (default), inches, centimeters or millimeters.

Parameters

- **x** (*point_like, rect_like*) – to which to compute the distance.
- **unit** (*str*) – the unit to be measured in. One of “px”, “in”, “cm”, “mm”.

Return type float

Returns

the distance to *x*. If this is *rect_like*, then the distance

- is the length of the shortest line connecting to one of the rectangle sides
- is calculated to the **finite version** of it
- is zero if it **contains** the point

norm ()

(*New in version 1.16.0*)

Return the Euclidean norm (the length) of the point as a vector. Equals result of function *abs()*.

transform (*m*)

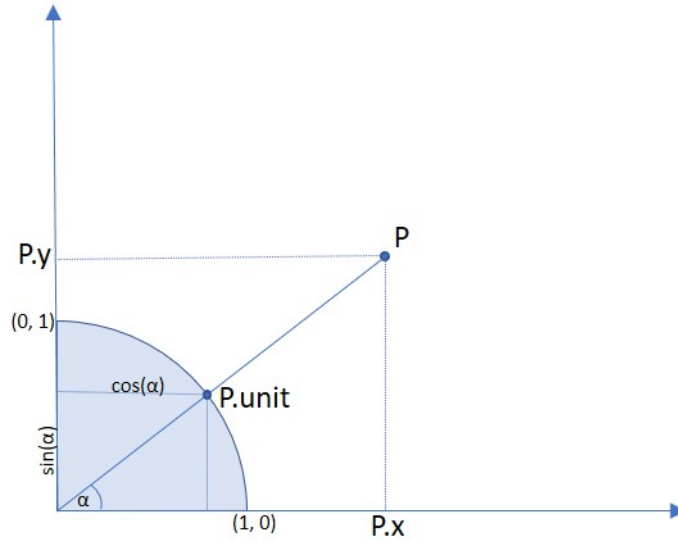
Apply a matrix to the point and replace it with the result.

Parameters **m** (*matrix_like*) – The matrix to be applied.

Return type *Point*

unit

Result of dividing each coordinate by *norm(point)*, the distance of the point to (0,0). This is a vector of length 1 pointing in the same direction as the point does. Its x, resp. y values are equal to the cosine, resp. sine of the angle this vector (and the point itself) has with the x axis.



Type *Point*

abs_unit

Same as *unit* above, replacing the coordinates with their absolute values.

Type *Point*

x

The x coordinate

Type float

y

The y coordinate

Type float

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
 - Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.
-

6.15 Quad

Represents a four-sided mathematical shape (also called “quadrilateral” or “tetragon”) in the plane, defined as a sequence of four *Point* objects ul, ur, ll, lr (conveniently called upper left, upper right, lower left, lower right).

Quads can **be obtained** as results of text search methods (*Page.searchFor()*), and they **are used** to define text marker annotations (see e.g. *Page.addSquigglyAnnot()* and friends), and in several draw methods (like *Page.drawQuad()* / *Shape.drawQuad()*, *Page.drawOval()* / :meth‘Shape.drawQuad’).

Note:

- If the corners of a rectangle are transformed with a **rotation**, **scale** or **translation** *Matrix*, then the resulting quad is **rectangular**, i.e. its corners again enclose angles of 90 degrees. Property *Quad.isRectangular* checks whether a quad can be thought of being the result of such an operation. This is not true for all matrices: e.g. shear matrices produce parallelograms, and non-invertible matrices deliver “degenerate” tetragons like triangles or lines.
- Attribute *Quad.rect* obtains the envelopping rectangle. Vice versa, rectangles now have attributes *Rect.quad*, resp. *IRect.quad* to obtain their respective tetragon versions.

Methods / Attributes	Short Description
<i>Quad.transform()</i>	transform with a matrix
<i>Quad.morph()</i>	transform with a point and matrix
<i>Quad.ul</i>	upper left point
<i>Quad.ur</i>	upper right point
<i>Quad.ll</i>	lower left point
<i>Quad.lr</i>	lower right point
<i>Quad.isConvex</i>	true if quad is a convex set
<i>Quad.isEmpty</i>	true if quad is an empty set
<i>Quad.isRectangular</i>	true if quad is a (rotated) rectangle
<i>Quad.rect</i>	smallest containing <i>Rect</i>
<i>Quad.width</i>	the longest width value
<i>Quad.height</i>	the longest height value

Class API

class Quad

`__init__(self)`

`__init__(self, ul, ur, ll, lr)`

`__init__(self, quad)`

`__init__(self, sequence)`

Overloaded constructors: “ul”, “ur”, “ll”, “lr” stand for *point_like* objects (the four corners), “sequence” is a Python sequence with four *point_like* objects.

If “quad” is specified, the constructor creates a **new copy** of it.

Without parameters, a quad consisting of 4 copies of *Point(0, 0)* is created.

transform (*matrix*)

Modify the quadrilateral by transforming each of its corners with a matrix.

Parameters *matrix* (*matrix_like*) – the matrix.

morph (*fixpoint*, *matrix*)

(*New in version 1.17.0*) “Morph” the quad with a matrix-like using a point-like as fixed point.

Parameters

- **fixpoint** (*point_like*) – the point.
- **matrix** (*matrix_like*) – the matrix.

Returns

a new quad. The effect is achieved by using the following code:

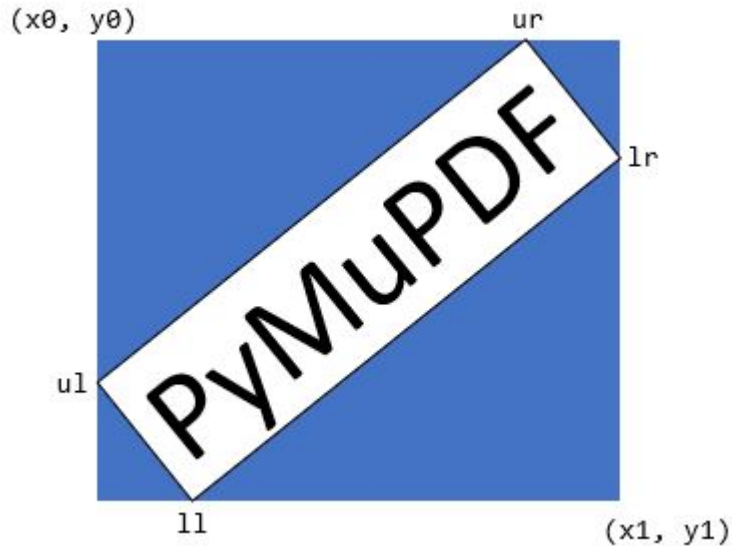
```
>>> T = fitz.Matrix(1, 1).preTranslate(fixpoint.x, fixpoint.y)
>>> result = self * ~T * matrix * T
```

So the quad is translated such, that fixpoint becomes the origin (0, 0), then the matrix is applied to it, and finally a reverse translation is done.

Typical uses include rotating the quad around a desired point.

rect

The smallest rectangle containing the quad, represented by the blue area in the following picture.



Type *Rect*

ul

Upper left point.

Type *Point*

ur

Upper right point.

Type *Point*

ll

Lower left point.

Type *Point*

lr

Lower right point.

Type *Point*

isConvex

(New in version 1.16.1)

True if every line connecting two points of the quad is inside the quad. We in addition also make sure here, that the quad is not “degenerate”, i.e. not all corners are on the same line (which would still qualify as convexity in the mathematical sense).

Type bool

isEmpty

True if enclosed area is zero, which means that at least three of the four corners are on the same line. If this is false, the quad may still be degenerate or not look like a tetragon at all (triangles, parallelograms, trapezoids, ...).

Type bool

isRectangular

True if all corner angles are 90 degrees. This implies that the quad is **convex and not empty**.

Type bool

width

The maximum length of the top and the bottom side.

Type float

height

The maximum length of the left and the right side.

Type float

6.15.1 Remark

This class adheres to the sequence protocol, so components can be dealt with via their indices, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.

We are still in process to extend algebraic operations to quads. Multiplication and division with / by numbers and matrices are already defined. Addition, subtraction and any unary operations may follow when we see an actual need.

6.16 Rect

Rect represents a rectangle defined by four floating point numbers x_0 , y_0 , x_1 , y_1 . They are treated as being coordinates of two diagonally opposite points. The first two numbers are regarded as the “top left” corner P_{x_0,y_0} and P_{x_1,y_1} 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: $\text{Rect}(P_{x_0,y_0}, P_{x_1,y_1})$, $\text{Rect}(P_{x_1,y_1}, P_{x_0,y_0})$, $\text{Rect}(P_{x_0,y_1}, P_{x_1,y_0})$, and $\text{Rect}(P_{x_1,y_0}, P_{x_0,y_1})$.

Hence some useful classification:

- A rectangle is called **finite** if $x_0 \leq x_1$ and $y_0 \leq y_1$ (i.e. the bottom right point is “south-eastern” to the top left one), otherwise **infinite**. Of the four alternatives above, **only one** is finite (disregarding degenerate cases). Please take into account, that in MuPDF’s coordinate system the y-axis is oriented from **top to bottom**.
- A rectangle is called **empty** if $x_0 = x_1$ or $y_0 = y_1$, 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
<code>Rect.contains()</code>	checks containment of another object
<code>Rect.getArea()</code>	calculate rectangle area
<code>Rect.getRectArea()</code>	calculate rectangle area
<code>Rect.includePoint()</code>	enlarge rectangle to also contain a point
<code>Rect.includeRect()</code>	enlarge rectangle to also contain another one
<code>Rect.intersect()</code>	common part with another rectangle
<code>Rect.intersects()</code>	checks for non-empty intersections
<code>Rect.morph()</code>	transform with a point and a matrix
<code>Rect.norm()</code>	the Euclidean norm
<code>Rect.normalize()</code>	makes a rectangle finite
<code>Rect.round()</code>	create smallest <i>IRect</i> containing rectangle
<code>Rect.transform()</code>	transform rectangle with a matrix
<code>Rect.bottom_left</code>	bottom left point, synonym <i>bl</i>
<code>Rect.bottom_right</code>	bottom right point, synonym <i>br</i>
<code>Rect.height</code>	rectangle height
<code>Rect.irect</code>	equals result of method <i>round()</i>
<code>Rect.isEmpty</code>	whether rectangle is empty
<code>Rect.isInfinite</code>	whether rectangle is infinite
<code>Rect.top_left</code>	top left point, synonym <i>tl</i>
<code>Rect.top_right</code>	top right point, synonym <i>tr</i>
<code>Rect.quad</code>	<i>Quad</i> made from rectangle corners
<code>Rect.width</code>	rectangle width
<code>Rect.x0</code>	top left corner's X-coordinate
<code>Rect.x1</code>	bottom right corner's X-coordinate
<code>Rect.y0</code>	top left corner's Y-coordinate
<code>Rect.y1</code>	bottom right corner's Y-coordinate

Class API

class Rect

`__init__(self)`

`__init__(self, x0, y0, x1, y1)`

`__init__(self, top_left, bottom_right)`

`__init__(self, top_left, x1, y1)`

`__init__(self, x0, y0, bottom_right)`

`__init__(self, rect)`

`__init__(self, sequence)`

Overloaded constructors: *top_left*, *bottom_right* stand for *point_like* objects, “sequence” is a Python sequence type of 4 numbers (see *Using Python Sequences as Arguments in PyMuPDF*), “rect” means another *rect_like*, while the other parameters mean coordinates.

If “rect” is specified, the constructor creates a **new copy** of it.

Without parameters, the empty rectangle *Rect(0.0, 0.0, 0.0, 0.0)* is created.

round()

Creates the smallest containing *IRect*. This is **not** the same as simply rounding the rectangle's edges: The top left corner is rounded upwards and left while the bottom right corner is rounded downwards and to the right.

```
>>> fitz.Rect(0.5, -0.01, 123.88, 455.123456).round()
IRect(0, -1, 124, 456)
```

1. If the rectangle is **infinite**, the “normalized” (finite) version of it will be taken. The result of this method is always a finite *IRect*.
2. If the rectangle is **empty**, the result is also empty.
3. **Possible paradox:** The result may be empty, **even if** the rectangle is **not** empty! In such cases, the result obviously does **not** contain the rectangle. This is because MuPDF’s algorithm allows for a small tolerance (1e-3). Example:

```
>>> r = fitz.Rect(100, 100, 200, 100.001)
>>> r.isEmpty # rect is NOT empty
False
>>> r.round() # but its irect IS empty!
fitz.IRect(100, 100, 200, 100)
>>> r.round().isEmpty
True
```

Return type *IRect*

transform (*m*)

Transforms the rectangle with a matrix and **replaces the original**. If the rectangle is empty or infinite, this is a no-operation.

Parameters *m* (*Matrix*) – The matrix for the transformation.

Return type *Rect*

Returns the smallest rectangle that contains the transformed original.

intersect (*r*)

The intersection (common rectangular area) of the current rectangle and *r* is calculated and **replaces the current** rectangle. If either rectangle is empty, the result is also empty. If *r* is infinite, this is a no-operation.

Parameters *r* (*Rect*) – Second rectangle

includeRect (*r*)

The smallest rectangle containing the current one and *r* is calculated and **replaces the current** one. If either rectangle is infinite, the result is also infinite. If one is empty, the other one will be taken as the result.

Parameters *r* (*Rect*) – Second rectangle

includePoint (*p*)

The smallest rectangle containing the current one and point *p* is calculated and **replaces the current** one. **Infinite rectangles remain unchanged.** To create a rectangle containing a series of points, start with (the empty) *fitz.Rect(p1, p1)* and successively perform *includePoint* operations for the other points.

Parameters *p* (*Point*) – Point to include.

getRectArea ([*unit*])

getArea ([*unit*])

Calculate the area of the rectangle and, with no parameter, equals *abs(rect)*. Like an empty rectangle, the area of an infinite rectangle is also zero. So, at least one of *fitz.Rect(p1, p2)* and *fitz.Rect(p2, p1)* has a zero area.

Parameters **unit** (*str*) – Specify required unit: respective squares of *px* (pixels, default), *in* (inches), *cm* (centimeters), or *mm* (millimeters).

Return type float

contains (*x*)

Checks whether *x* is contained in the rectangle. It may be an *IRect*, *Rect*, *Point* or number. If *x* is an empty rectangle, this is always true. If the rectangle is empty this is always *False* for all non-empty rectangles and for all points. If *x* is a number, it will be checked against the four components. *x in rect* and *rect.contains(x)* are equivalent.

Parameters **x** (*IRect* or *Rect* or *Point* or number) – the object to check.

Return type bool

intersects (*r*)

Checks whether the rectangle and a *rect_like* “*r*” contain a common non-empty *Rect*. This will always be *False* if either is infinite or empty.

Parameters **r** (*rect_like*) – the rectangle to check.

Return type bool

morph (*fixpoint*, *matrix*)

(New in version 1.17.0)

Return a new quad after applying a matrix to it using a fixed point.

Parameters

- **fixpoint** (*point_like*) – the fixed point.
- **matrix** (*matrix_like*) – the matrix.

Returns a new *Quad*. This a wrapper for the same-named quad method.

norm ()

(New in version 1.16.0)

Return the Euclidean norm of the rectangle treated as a vector of four numbers.

normalize ()

Replace the rectangle with its finite version. This is done by shuffling the rectangle corners. After completion of this method, the bottom right corner will indeed be south-eastern to the top left one.

irect

Equals result of method *round*().

top_left

t1

Equals *Point(x0, y0)*.

Type *Point*

top_right

tr

Equals *Point(x1, y0)*.

Type *Point*

bottom_left

b1

Equals *Point(x0, y1)*.

Type *Point*

bottom_right

br

Equals *Point(x1, y1)*.

Type *Point*

quad

The quadrilateral *Quad(rect.tl, rect.tr, rect.bl, rect.br)*.

Type *Quad*

width

Width of the rectangle. Equals *abs(x1 - x0)*.

Return type float

height

Height of the rectangle. Equals *abs(y1 - y0)*.

Return type float

x0

X-coordinate of the left corners.

Type float

y0

Y-coordinate of the top corners.

Type float

x1

X-coordinate of the right corners.

Type float

y1

Y-coordinate of the bottom corners.

Type float

isInfinite

True if rectangle is infinite, *False* otherwise.

Type bool

isEmpty

True if rectangle is empty, *False* otherwise.

Type bool

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
 - Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.
-

6.17 Shape

This class allows creating interconnected graphical elements on a PDF page. Its methods have the same meaning and name as the corresponding *Page* methods.

In fact, each *Page* draw method is just a convenience wrapper for (1) one shape draw method, (2) the `finish()` method, and (3) the `commit()` method. For page text insertion, only the `commit()` method is invoked. If many draw and text operations are executed for a page, you should always consider using a *Shape* object.

Several draw methods can be executed in a row and each one of them will contribute to one drawing. Once the drawing is complete, the `finish()` method must be invoked to apply color, dashing, width, morphing and other attributes.

Draw methods of this class (and `insertTextbox()`) are logging the area they are covering in a rectangle (*Shape.rect*). This property can for instance be used to set *Page.CropBox*.

Text insertions `insertText()` and `insertTextbox()` implicitly execute a “finish” and therefore only require `commit()` to become effective. As a consequence, both include parameters for controlling properties like colors, etc.

Method / Attribute	Description
<i>Shape.commit()</i>	update the page's contents
<i>Shape.drawBezier()</i>	draw a cubic Bezier curve
<i>Shape.drawCircle()</i>	draw a circle around a point
<i>Shape.drawCurve()</i>	draw a cubic Bezier using one helper point
<i>Shape.drawLine()</i>	draw a line
<i>Shape.drawOval()</i>	draw an ellipse
<i>Shape.drawPolyline()</i>	connect a sequence of points
<i>Shape.drawQuad()</i>	draw a quadrilateral
<i>Shape.drawRect()</i>	draw a rectangle
<i>Shape.drawSector()</i>	draw a circular sector or piece of pie
<i>Shape.drawSquiggle()</i>	draw a squiggly line
<i>Shape.drawZigzag()</i>	draw a zigzag line
<i>Shape.finish()</i>	finish a set of draw commands
<i>Shape.insertText()</i>	insert text lines
<i>Shape.insertTextbox()</i>	fit text into a rectangle
<i>Shape.doc</i>	stores the page's document
<i>Shape.draw_cont</i>	draw commands since last <i>finish()</i>
<i>Shape.height</i>	stores the page's height
<i>Shape.lastPoint</i>	stores the current point
<i>Shape.page</i>	stores the owning page
<i>Shape.rect</i>	rectangle surrounding drawings
<i>Shape.text_cont</i>	accumulated text insertions
<i>Shape.totalcont</i>	accumulated string to be stored in <i>contents</i>
<i>Shape.width</i>	stores the page's width

Class API

class Shape

`__init__(self, page)`

Create a new drawing. During importing PyMuPDF, the *fitz.Page* object is being given the convenience method *newShape()* to construct a *Shape* object. During instantiation, a check will be made whether we do have a PDF page. An exception is otherwise raised.

Parameters *page* (*Page*) – an existing page of a PDF document.

drawLine (*p1*, *p2*)

Draw a line from *point_like* objects *p1* to *p2*.

Parameters

- **p1** (*point_like*) – starting point
- **p2** (*point_like*) – end point

Return type *Point*

Returns the end point, *p2*.

drawSquiggle (*p1*, *p2*, *breadth*=2)

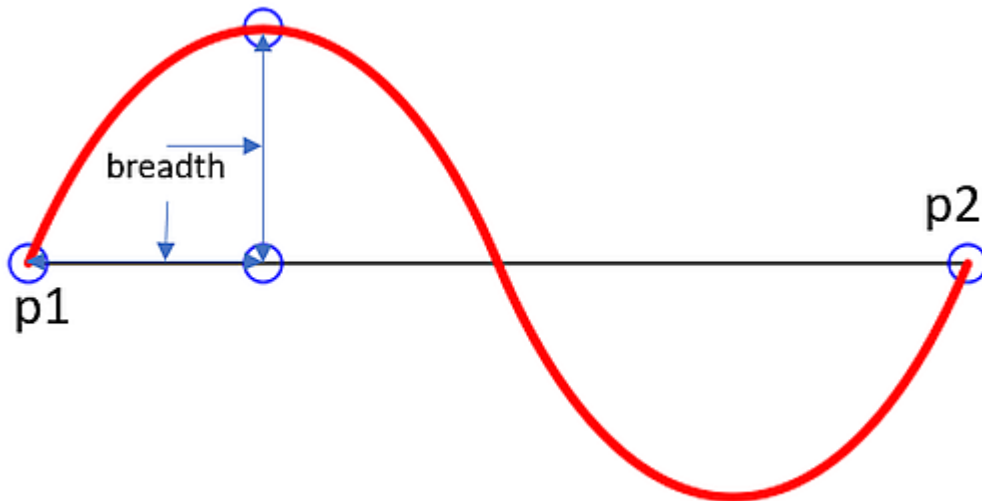
Draw a squiggly (wavy, undulated) line from *point_like* objects *p1* to *p2*. An integer number of full wave periods will always be drawn, one period having a length of $4 * \textit{breadth}$. The *breadth* parameter will be adjusted as necessary to meet this condition. The drawn line will always turn “left” when leaving *p1* and always join *p2* from the “right”.

Parameters

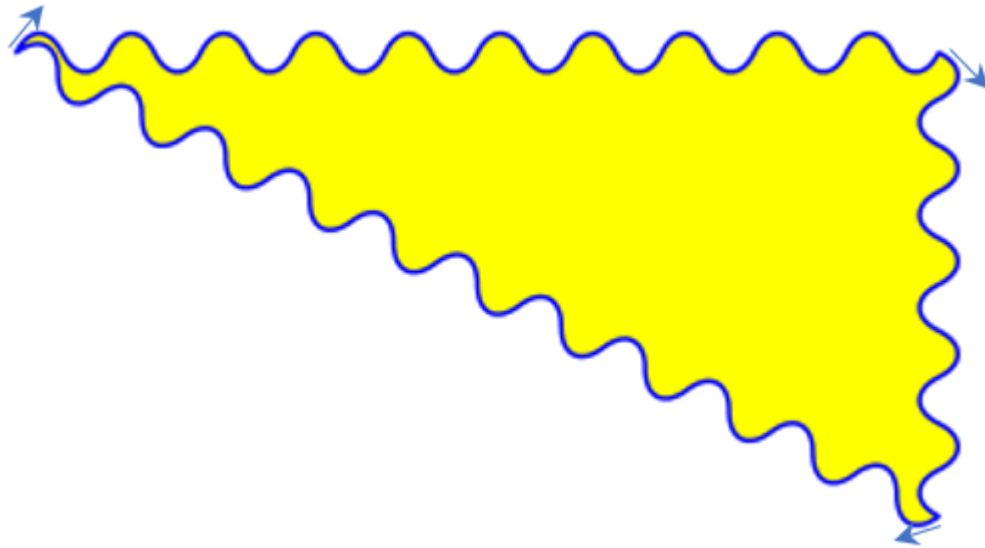
- **p1** (*point_like*) – starting point
- **p2** (*point_like*) – end point
- **breadth** (*float*) – the amplitude of each wave. The condition $2 * \textit{breadth} < \textit{abs}(p2 - p1)$ must be true to fit in at least one wave. See the following picture, which shows two points connected by one 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_like* objects *p1* to *p2*. An integer number of full zigzag periods will always be drawn, one period having a length of $4 * \textit{breadth}$. The *breadth* parameter will be adjusted to meet this condition. The drawn line will always turn “left” when leaving *p1* and always join *p2* from the “right”.

Parameters

- **p1** (*point_like*) – starting point
- **p2** (*point_like*) – end point
- **breadth** (*float*) – the amplitude of the movement. The condition $2 * \textit{breadth} < \textit{abs}(p2 - p1)$ must be true to fit in at least one period.

Return type *Point*

Returns the end point, *p2*.

drawPolyline (*points*)

Draw several connected lines between points contained in the sequence *points*. This can be used for creating arbitrary polygons by setting the last item equal to the first one.

Parameters **points** (*sequence*) – a sequence of *point_like* objects. Its length must at least be 2 (in which case it is equivalent to *drawLine()*).

Return type *Point*

Returns *points[-1]* – the last point in the argument sequence.

drawBezier (*p1*, *p2*, *p3*, *p4*)

Draw a standard cubic Bezier curve from *p1* to *p4*, using *p2* and *p3* as control points.

All arguments are *point_like* s.

Return type *Point*

Returns the end point, *p4*.

Note: The points do not need to be different – experiment a bit with some of them being equal!

Example:



drawOval (*tetra*)

Draw an “ellipse” inside the given tetragon (quadrilateral). If it is a square, a regular circle is drawn, a general rectangle will result in an ellipse. If a quadrilateral is used instead, a plethora of shapes can be the result.

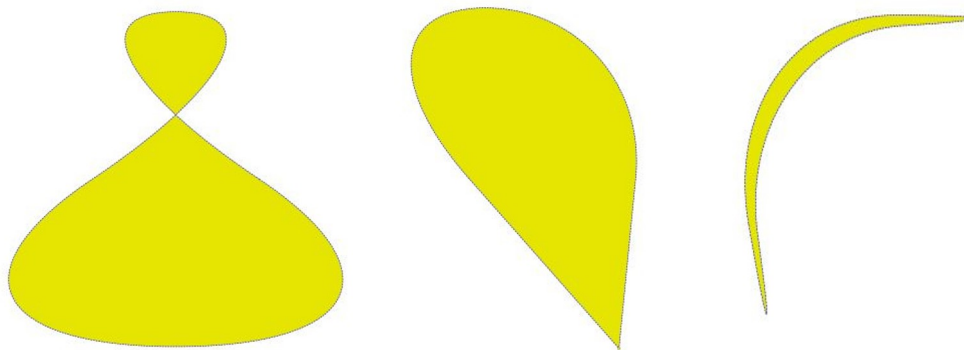
The drawing starts and ends at the middle point of the line connecting bottom-left and top-left corners in an anti-clockwise movement.

Parameters **tetra** (*rect_like*, *quad_like*) – *rect_like* or *quad_like*.

Changed in version 1.14.5: tetragons are now also supported.

Return type *Point*

Returns the middle point of line from *rect.bl* to *rect.tl*, or from *quad.ll* to *quad.ul*, respectively. Look at just a few examples here, or at the *quad-show?.py* scripts in the PyMuPDF-Utilities repository.



drawCircle (*center*, *radius*)

Draw a circle given its center and radius. The drawing starts and ends at point *center* - (*radius*, 0) in an anti-clockwise movement. This corresponds to the middle point of the enclosing rectangle’s left side.

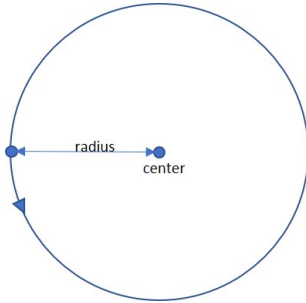
The method is a shortcut for *drawSector(center, start, 360, fullSector=False)*. To draw a circle in a clockwise movement, change the sign of the degree.

Parameters

- **center** (*point_like*) – the center of the circle.
- **radius** (*float*) – the radius of the circle. Must be positive.

Return type *Point*

Returns $center - (radius, 0)$.



drawCurve (*p1, p2, p3*)

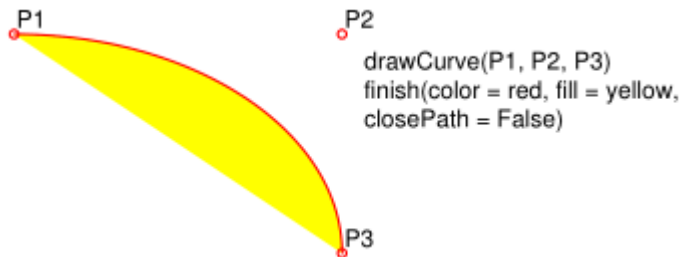
A special case of *drawBezier()*: Draw a cubic Bezier curve from *p1* to *p3*. On each of the two lines from *p1* to *p2* and from *p2* to *p3* one control point is generated. This guaranties that the curve's curvature does not change its sign. If these two connecting lines intersect with an angle of 90 degrees, then the resulting curve is a quarter ellipse (or quarter circle, if of same length) circumference.

All arguments are *point_like*.

Return type *Point*

Returns the end point, *p3*.

Example: a filled quarter ellipse segment.



drawSector (*center, point, angle, fullSector=True*)

Draw a circular sector, optionally connecting the arc to the circle's center (like a piece of pie).

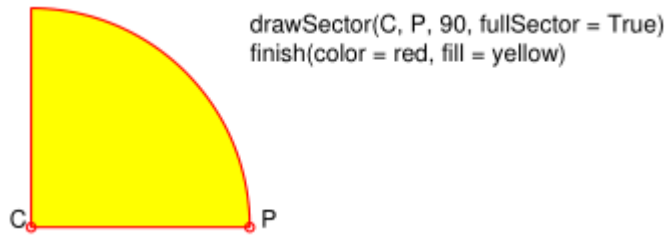
Parameters

- **center** (*point_like*) – the center of the circle.
- **point** (*point_like*) – one of the two end points of the pie's arc segment. The other one is calculated from the *angle*.
- **angle** (*float*) – the angle of the sector in degrees. Used to calculate the other end point of the arc. Depending on its sign, the arc is drawn anti-clockwise (postive) or clockwise.
- **fullSector** (*bool*) – whether to draw connecting lines from the ends of the arc to the circle center. If a fill color is specified, the full “pie” is colored, otherwise just the sector.

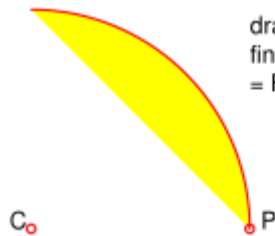
Returns the other end point of the arc. Can be used as starting point for a following invocation to create logically connected pies charts.

Return type *Point*

Examples:



```
drawSector(C, P, 90, fullSector = True)
finish(color = red, fill = yellow)
```



```
drawSector(C, P, 90, fullSector = False)
finish(color = red, fill = yellow, closePath
= False)
```

drawRect (*rect*)

Draw a rectangle. The drawing starts and ends at the top-left corner in an anti-clockwise movement.

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

Return type *Point*

Returns top-left corner of the rectangle.

drawQuad (*quad*)

Draw a quadrilateral. The drawing starts and ends at the top-left corner (*Quad.ul*) in an anti-clockwise movement. It invokes *drawPolyline()* with the argument [*ul, ll, lr, ur, ul*].

Parameters *quad* (*quad_like*) – where to put the tetragon on the page.

Return type *Point*

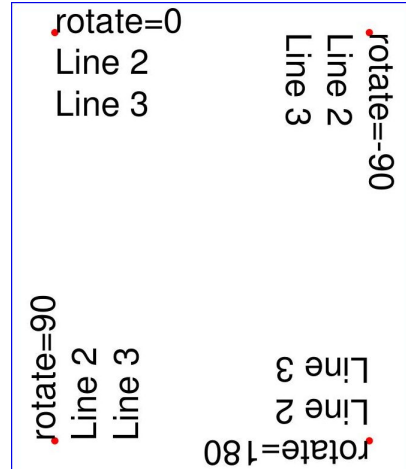
Returns *Quad.ul*.

insertText (*point, text, fontsize=11, fontname="helv", fontfile=None, set_simple=False, encoding=TEXT_ENCODING_LATIN, color=None, fill=None, render_mode=0, border_width=1, rotate=0, morph=None, stroke_opacity=1, fill_opacity=1, oc=0*)

Insert text lines start at *point*.

Parameters

- **point** (*point_like*) – the bottom-left position of the first character of *text* in pixels. It is important to understand, how this works in conjunction with the *rotate* parameter. Please have a look at the following picture. The small red dots indicate the posi-



tions of *point* in each of the four possible cases.

- **text** (*str/sequence*) – the text to be inserted. May be specified as either a string type or as a sequence type. For sequences, or strings containing line breaks *n*, several lines will be inserted. No care will be taken if lines are too wide, but the number of inserted lines will be limited by “vertical” space on the page (in the sense of reading direction as established by the *rotate* parameter). Any rest of *text* is discarded – the return code however contains the number of inserted lines.
- **stroke_opacity** (*float*) – (*new in v1.18.1*) set transparency for stroke colors. Negative values and values > 1 will be ignored. Default is 1 (intransparent).
- **fill_opacity** (*float*) – (*new in v1.18.1*) set transparency for fill colors. Default is 1 (intransparent). Use this value to control transparency of the text color. Stroke opacity **only** affects the border line of characters.
- **rotate** (*int*) – determines whether to rotate the text. Acceptable values are multiples of 90 degrees. Default is 0 (no rotation), meaning horizontal text lines oriented from left to right. 180 means text is shown upside down from **right to left**. 90 means anti-clockwise rotation, text running **upwards**. 270 (or -90) means clockwise rotation, text running **downwards**. In any case, *point* specifies the bottom-left coordinates of the first character’s rectangle. Multiple lines, if present, always follow the reading direction established by this parameter. So line 2 is located **above** line 1 in case of *rotate = 180*, etc.
- **oc** (*int*) – (*new in v1.18.4*) the *xref* number of an *OCG* or *OCMD* to make this text conditionally displayable.

Return type int

Returns number of lines inserted.

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

insertTextbox (*rect, buffer, fontsize=11, fontname="helv", fontfile=None, set_simple=False, encoding=TEXT_ENCODING_LATIN, color=None, fill=None, render_mode=0, border_width=1, expandtabs=8, align=TEXT_ALIGN_LEFT, rotate=0, morph=None, stroke_opacity=1, fill_opacity=1, oc=0*)

PDF only: Insert text into the specified rectangle. The text will be split into lines and words and then filled into the available space, starting from one of the four rectangle corners, which depends on *rotate*. Line feeds will be respected as well as multiple spaces will be.

Parameters

- **rect** (*rect_like*) – the area to use. It must be finite and not empty.

- **buffer** (*str/sequence*) – the text to be inserted. Must be specified as a string or a sequence of strings. Line breaks are respected also when occurring in a sequence entry.
- **align** (*int*) – align each text line. Default is 0 (left). Centered, right and justified are the other supported options, see *Text Alignment*. Please note that the effect of parameter value `TEXT_ALIGN_JUSTIFY` is only achievable with “simple” (single-byte) fonts (including the *PDF Base 14 Fonts*). Refer to *Adobe PDF References*, section 5.2.2, page 399.
- **expandtabs** (*int*) – controls handling of tab characters *t* using the `string.expandtabs()` method **per each line**.
- **stroke_opacity** (*float*) – (*new in v1.18.1*) set transparency for stroke colors. Negative values and values > 1 will be ignored. Default is 1 (intransparent).
- **fill_opacity** (*float*) – (*new in v1.18.1*) set transparency for fill colors. Default is 1 (intransparent). Use this value to control transparency of the text color. Stroke opacity **only** affects the border line of characters.
- **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 its lower left corner.
- **oc** (*int*) – (*new in v1.18.4*) the *xref* number of an *OCG* or *OCMD* to make this text conditionally displayable.

Return type float

Returns

If positive or zero: successful execution. The value returned is the unused rectangle line space in pixels. This may safely be ignored – or be used to optimize the rectangle, position subsequent items, etc.

If negative: no execution. The value returned is the space deficit to store text lines. Enlarge rectangle, decrease *fontsize*, decrease text amount, etc.



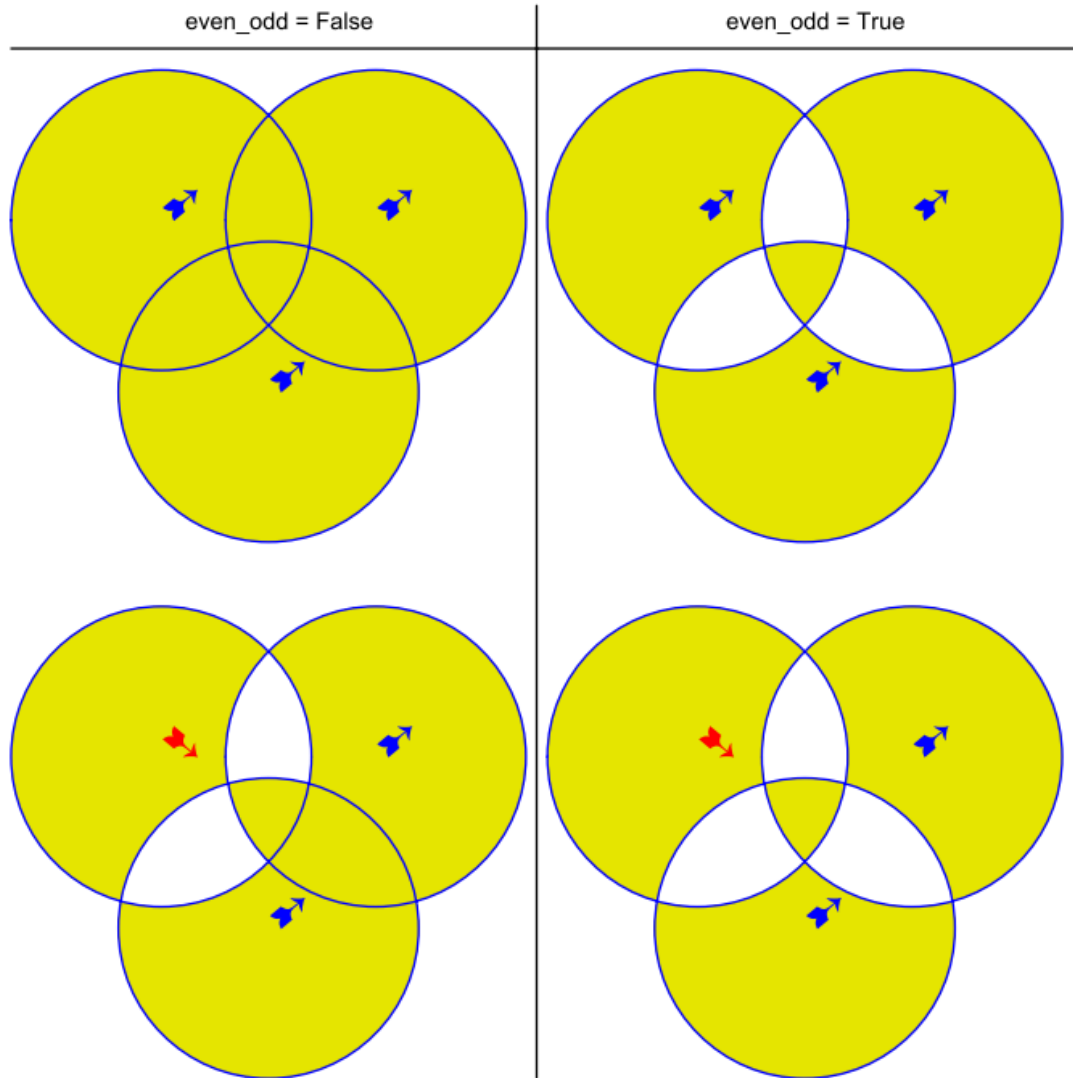


For a description of the other parameters see *Common Parameters*.

finish (*width=1, color=None, fill=None, lineCap=0, lineJoin=0, dashes=None, closePath=True, even_odd=False, morph=(fixpoint, matrix), stroke_opacity=1, fill_opacity=1, oc=0*)
 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 fixpoint *Point*.

Parameters

- **morph** (*sequence*) – morph the text or the compound drawing around some arbitrary *Point fixpoint* by applying *Matrix matrix* to it. This implies that *fixpoint* is a **fixed point** of this operation: it will not change its position. 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.
- **stroke_opacity** (*float*) – (*new in v1.18.1*) set transparency for stroke colors. Value < 0 or > 1 will be ignored. Default is 1 (intransparent).
- **fill_opacity** (*float*) – (*new in v1.18.1*) set transparency for fill colors. Default is 1 (intransparent).
- **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 References*, pp. 232 ff. Here is an example to demonstrate the difference.
- **oc** (*int*) – (*new in v1.18.4*) the *xref* number of an *OCG* or *OCMD* to make this drawing conditionally displayable.



Note: For each pixel in a drawing the following will happen:

1. Rule “**even-odd**” counts, how many areas are overlapping at a pixel. If this count is **odd** the pixel is regarded **inside**, if it is **even**, the pixel is **outside**.
2. Default rule “**nonzero winding**” also looks at the orientation of overlapping areas: it **adds 1** if an area is drawn anti-clockwise and it **subtracts 1** for clockwise areas. If the result is zero, the pixel is regarded **outside**, pixels with a non-zero count are **inside**.

In the top two shapes, three circles are drawn in standard manner (anti-clockwise, look at the arrows). The lower two shapes contain one (top-left) circle drawn clockwise. As can be seen, area orientation is irrelevant for the even-odd rule.

commit (*overlay=True*)

Update the page’s *contents* with the accumulated draw commands and text insertions. If a *Shape* is not committed, the page will not be changed.

The method will reset attributes *Shape.rect*, *lastPoint*, *draw_cont*, *text_cont* and *totalcont*. Afterwards, the shape object can be reused for the **same page**.

Parameters `overlay` (*bool*) – determine whether to put content in foreground (default) or background. Relevant only, if the page already has a non-empty `contents` object.

doc

For reference only: the page’s document.

Type *Document*

page

For reference only: the owning page.

Type *Page*

height

Copy of the page’s height

Type float

width

Copy of the page’s width.

Type float

draw_cont

Accumulated command buffer for **draw methods** since last finish.

Type str

text_cont

Accumulated text buffer. All **text insertions** go here. On `commit()` this buffer will be appended to `totalcont`, so that text will never be covered by drawings in the same Shape.

Type str

rect

Rectangle surrounding drawings. This attribute is at your disposal and may be changed at any time. Its value is set to *None* when a shape is created or committed. Every `draw*` method, and `Shape.insertTextbox()` update this property (i.e. **enlarge** the rectangle as needed). **Morphing** operations, however (`Shape.finish()`, `Shape.insertTextbox()`) are ignored.

A typical use of this attribute would be setting `Page.CropBox` to this value, when you are creating shapes for later or external use. If you have not manipulated the attribute yourself, it should reflect a rectangle that contains all drawings so far.

If you have used morphing and need a rectangle containing the morphed objects, use the following code:

```
>>> # assuming ...
>>> morph = (point, matrix)
>>> # ... recalculate the shape rectangle like so:
>>> shape.rect = (shape.rect - fitz.Rect(point, point)) * ~matrix + fitz.
↳Rect(point, point)
```

Type *Rect*

totalcont

Total accumulated command buffer for draws and text insertions. This will be used by `Shape.commit()`.

Type str

lastPoint

For reference only: the current point of the drawing path. It is *None* at `Shape` creation and after each `finish()` and `commit()`.

Type *Point*

6.17.1 Usage

A drawing object is constructed by `shape = page.newShape()`. After this, as many `draw`, `finish` and `text` insertions methods as required may follow. Each sequence of draws must be finished before the drawing is committed. The overall coding pattern looks like this:

```
>>> shape = page.newShape()
>>> shape.draw1(...)
>>> shape.draw2(...)
>>> ...
>>> shape.finish(width=..., color=..., fill=..., morph=...)
>>> shape.draw3(...)
>>> shape.draw4(...)
>>> ...
>>> shape.finish(width=..., color=..., fill=..., morph=...)
>>> ...
>>> shape.insertText*
>>> ...
>>> shape.commit()
>>> .....
```

Note:

1. Each `finish()` combines the preceding draws into one logical shape, giving it common colors, line width, morphing, etc. If `closePath` is specified, it will also connect the end point of the last draw with the starting point of the first one.
2. To successfully create compound graphics, let each draw method use the end point of the previous one as its starting point. In the above pseudo code, `draw2` should hence use the returned *Point* of `draw1` as its starting point. Failing to do so, would automatically start a new path and `finish()` may not work as expected (but it won't complain either).
3. Text insertions may occur anywhere before the commit (they neither touch `Shape.draw_cont` nor `Shape.lastPoint`). They are appended to `Shape.totalcont` directly, whereas draws will be appended by `Shape.finish`.
4. Each `commit` takes all text insertions and shapes and places them in foreground or background on the page – thus providing a way to control graphical layers.
5. **Only `commit` will update** the page's contents, the other methods are basically string manipulations.

6.17.2 Examples

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

```
shape = page.newShape() # start a new shape
cols = (...) # a sequence of RGB color triples
pieces = len(cols) # number of pieces to draw
beta = 360. / pieces # angle of each piece of pie
center = fitz.Point(...) # center of the pie
p0 = fitz.Point(...) # starting point
for i in range(pieces):
    p0 = shape.drawSector(center, p0, beta,
```

(continues on next page)

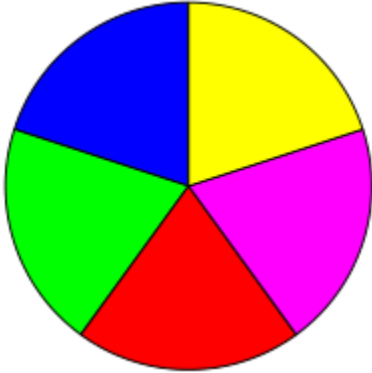
(continued from previous page)

```

                                fullSector=True) # draw piece
    # now fill it but do not connect ends of the arc
    shape.finish(fill=cols[i], closePath=False)
shape.commit() # update the page

```

Here is an example for 5 colors:



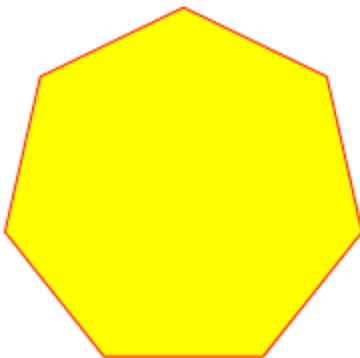
2. Create a regular n-edged polygon (fill yellow, red border). We use `drawSector()` only to calculate the points on the circumference, and empty the draw command buffer again before drawing the polygon:

```

shape = page.newShape() # start a new shape
beta = -360.0 / n # our angle, drawn clockwise
center = fitz.Point(...) # center of circle
p0 = fitz.Point(...) # start here (1st edge)
points = [p0] # store polygon edges
for i in range(n): # calculate the edges
    p0 = shape.drawSector(center, p0, beta)
    points.append(p0)
shape.draw_cont = "" # do not draw the circle sectors
shape.drawPolyline(points) # draw the polygon
shape.finish(color=(1,0,0), fill=(1,1,0), closePath=False)
shape.commit()

```

Here is the polygon for $n = 7$:



6.17.3 Common Parameters

`fontname` (*str*)

In general, there are three options:

1. Use one of the standard *PDF Base 14 Fonts*. In this case, *fontfile* **must not** be specified and “*Helvetica*” is used if this parameter is omitted, too.
2. Choose a font already in use by the page. Then specify its **reference** name prefixed with a slash “/”, see example below.
3. Specify a font file present on your system. In this case choose an arbitrary, but new name for this parameter (without “/” prefix).

If inserted text should re-use one of the page’s fonts, use its reference name appearing in `getFontList()` like so:

Suppose the font list has the entry `[1024, 0, 'Type1', 'CJXQIC+NimbusMonL-Bold', 'R366']`, then specify `fontname = "/R366"`, `fontfile = None` to use font *CJXQIC+NimbusMonL-Bold*.

fontfile (*str*)

File path of a font existing on your computer. If you specify *fontfile*, make sure you use a *fontname* **not occurring** in the above list. This new font will be embedded in the PDF upon `doc.save()`. Similar to new images, a font file will be embedded only once. A table of MD5 codes for the binary font contents is used to ensure this.

set_simple (*bool*)

Fonts installed from files are installed as **Type0** fonts by default. If you want to use 1-byte characters only, set this to true. This setting cannot be reverted. Subsequent changes are ignored.

fontsize (*float*)

Font size of text. This also determines the line height as `fontsize * 1.2`.

dashes (*str*)

Causes lines to be drawn dashed. The general format is “`[n m] p`”. The square brackets denote a PDF array of one or two floats. Float *p* is called the “dash phase” and specifies how many pixels should be skipped before the dashing starts.

A continuous line (no dashes) is drawn with “[*l*] 0” or *None* or “”. Specifying “[*3 4*] 0” means dashes of 3 and gaps of 4 pixels following each other. “[*3 3*] 0” and “[*3*] 0” do the same thing. For (the rather complex) details on how to achieve sophisticated dashing effects, see *Adobe PDF References*, page 217.

color / fill (*list, tuple*)

Line and fill colors can be specified as tuples or list of floats from 0 to 1. These sequences must have a length of 1 (GRAY), 3 (RGB) or 4 (CMYK). For GRAY colorspace, a single float instead of the unwieldy (*float*,) or [*float*] is also accepted.

To simplify color specification, method `getColor()` in `fitz.utils` may be used to get predefined RGB color triples by name. It accepts a string as the name of the color and returns the corresponding triple. The method knows over 540 color names – see section *Color Database*.

Please note that the term *color* usually means “stroke” color when used in conjunction with fill color.

stroke_opacity / fill_opacity (*floats*)

Both values are floats in range [0, 1]. Negative values or values > 1 will ignored (in most cases). Both set the transparency such that a value 0.5 corresponds to 50% transparency, 0 means invisible and 1 means intransparent. For e.g. a rectangle the stroke opacity applies to its border and fill opacity to its interior.

For text insertions (*Shape.insertText()* and *Shape.insertTextbox()*), use *fill_opacity* for the text. At first sight this seems surprising, but it becomes obvious when you look further down to *render_mode*: *fill_opacity* applies to the yellow and *stroke_opacity* applies to the blue color.

border_width (*float*)

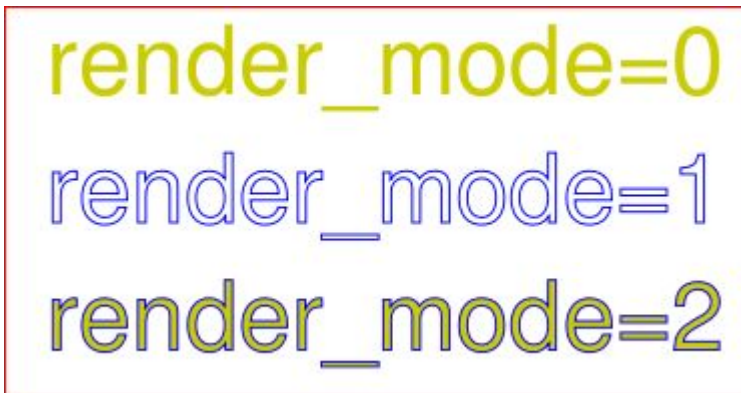
Set the border width for text insertions. New in v1.14.9. Relevant only if the render mode argument is used with a value greater zero.

render_mode (*int*)

New in version 1.14.9: Integer in *range(8)* which controls the text appearance (*Shape.insertText()* and *Shape.insertTextbox()*). See page 398 in *Adobe PDF References*. New in v1.14.9. These methods now also differentiate between fill and stroke colors.

- For default 0, only the text fill color is used to paint the text. For backward compatibility, using the *color* parameter instead also works.
- For render mode 1, only the border of each glyph (i.e. text character) is drawn with a thickness as set in argument *border_width*. The color chosen in the *color* argument is taken for this, the *fill* parameter is ignored.
- For render mode 2, the glyphs are filled and stroked, using both color parameters and the specified border width. You can use this value to simulate **bold text** without using another font: choose the same value for *fill* and *color* and an appropriate value for *border_width*.
- For render mode 3, the glyphs are neither stroked nor filled: the text becomes invisible.

The following examples use *border_width=0.3*, together with a *fontsize* of 15. Stroke color is blue and fill color is some yellow.



overlay (*bool*)

Causes the item to appear in foreground (default) or background.

morph (*sequence*)

Causes “morphing” of either a shape, created by the *draw*()* methods, or the text inserted by page methods *insertTextbox()* / *insertText()*. If not *None*, it must be a pair (*fixpoint*, *matrix*), where *fixpoint* 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 fixed point for the matrix operation. For example, if *matrix* is a rotation or scaling, then *fixpoint* 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 *fixpoint*, etc.

Note: Several methods contain checks whether the to be inserted items will actually fit into the page (like *Shape.insertText()*, or *Shape.drawRect()*). For the result of a morphing operation there is however no such guaranty: this is entirely the rpogrammer’s responsibility.

lineCap (deprecated: “roundCap”) (*int*)

Controls the look of line ends. The default value 0 lets each line end at exactly the given coordinate in a sharp edge. A value of 1 adds a semi-circle to the ends, whose center is the end point and whose diameter is the line width. Value 2 adds a semi-square with an edge length of line width and a center of the line end.

Changed in version 1.14.15

lineJoin (*int*)

New in version 1.14.15: Controls the way how line connections look like. This may be either as a sharp edge (0), a rounded join (1), or a cut-off edge (2, “butt”).

closePath (*bool*)

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

6.18 TextPage

This class represents text and images shown on a document page. All MuPDF document types are supported.

The usual ways to create a textpage are *DisplayList.getTextPage()* and *Page.getTextPage()*. Because there is a limited set of methods in this class, there exist wrappers in the *Page* class, which incorporate creating an intermediate text page and then invoke one of the following methods. The last column of this table shows these corresponding *Page* methods.

For a description of what this class is all about, see Appendix 2.

Method	Description	page getText or search method
<code>extractText ()</code>	extract plain text	“text”
<code>extractTEXT ()</code>	synonym of previous	“text”
<code>extractBLOCKS ()</code>	plain text grouped in blocks	“blocks”
<code>extractWORDS ()</code>	all words with their bbox	“words”
<code>extractHTML ()</code>	page content in HTML format	“html”
<code>extractXHTML ()</code>	page content in XHTML format	“xhtml”
<code>extractXML ()</code>	page text in XML format	“xml”
<code>extractDICT ()</code>	page content in <i>dict</i> format	“dict”
<code>extractJSON ()</code>	page content in JSON format	“json”
<code>extractRAWDICT ()</code>	page content in <i>dict</i> format	“rawdict”
<code>extractRAWJSON ()</code>	page content in JSON format	“rawjson”
<code>search ()</code>	Search for a string in the page	<code>Page.search ()</code>

Class API

class `TextPage`

`extractText ()`

`extractTEXT ()`

Return a string of the page’s complete text. The text is UTF-8 unicode and in the same sequence as specified at the time of document creation.

Return type str

`extractBLOCKS ()`

Textpage content as a list of text lines grouped by block. Each list items looks like this:

```
(x0, y0, x1, y1, "lines in blocks", block_no, block_type)
```

The first four entries are the block’s bbox coordinates, *block_type* is 1 for an image block, 0 for text. *block_no* is the block sequence number.

For an image block, its bbox and a text line with image meta information is included – not the image data itself.

This is a high-speed method with just enough information to output plain text in desired reading sequence.

Return type list

`extractWORDS ()`

Textpage content as a list of single words with bbox information. An item of this list looks like this:

```
(x0, y0, x1, y1, "word", block_no, line_no, word_no)
```

Everything wrapped in spaces is treated as a “*word*” with this method.

This is a high-speed method which e.g. allows extracting text from within a given rectangle.

Return type list

`extractHTML ()`

Textpage content in HTML format. This version contains complete formatting and positioning information. Images are included (encoded as base64 strings). You need an HTML package to interpret the output in Python. Your internet browser should be able to adequately display this information, but see *Controlling Quality of HTML Output*.

Return type str

extractDICT ()

Textpage content as a Python dictionary. Provides same information detail as HTML. See below for the structure.

Return type dict

extractJSON ()

Textpage content in JSON format. Created by `json.dumps(TextPage.extractDICT())`. It is included for backlevel compatibility. You will probably use this method ever only for outputting the result to some file. The method detects binary image data and converts them to base64 encoded strings on JSON output.

Return type str

extractXHTML ()

Textpage content in XHTML format. Text information detail is comparable with `extractTEXT()`, but also contains images (base64 encoded). This method makes no attempt to re-create the original visual appearance.

Return type str

extractXML ()

Textpage content in XML format. This contains complete formatting information about every single character on the page: font, size, line, paragraph, location, color, etc. Contains no images. You probably need an XML package to interpret the output in Python.

Return type str

extractRAWdict ()

Textpage content as a Python dictionary – technically similar to `extractDICT()`, and it contains that information as a subset (including any images). It provides additional detail down to each character, which makes using XML obsolete in many cases. See below for the structure.

Return type dict

extractRAWJSON ()

Textpage content in JSON format. Created by `json.dumps(TextPage.extractRAWdict())`. You will probably use this method ever only for outputting the result to some file. The method detects binary image data and converts them to base64 encoded strings on JSON output.

Return type str

search (*needle*, *quads*=False)

(Changed in v1.18.2)

Search for *string* and return a list of found locations.

Parameters

- **needle** (*str*) – the string to search for. Upper and lower cases will all match.
- **quads** (*bool*) – return quadrilaterals instead of rectangles.

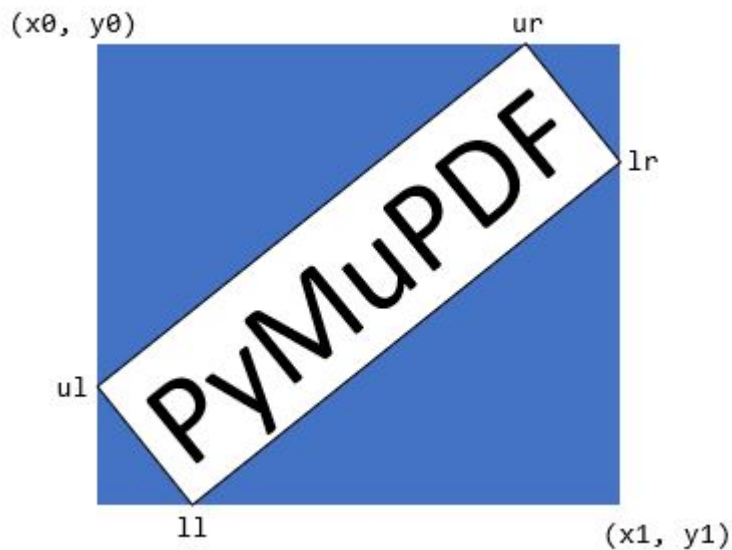
Return type list

Returns a list of *Rect* or *Quad* objects, each surrounding a found *needle* occurrence. The search string may contain spaces, it may therefore happen, that its parts are located on different lines. In this case, more than one rectangle (resp. quadrilateral) are returned. **(Changed in v1.18.2)** The method **now supports dehyphenation**, so it will find “method” even if it was hyphenated in two parts “meth-” and “od” across two lines. The two returned rectangles will **exclude the hyphen** in this case.

Note: Overview of changes in v1.18.2:

1. The `hit_max` parameter has been removed: all hits are always returned.
 2. The `rect` parameter of the `TextPage` is now respected: only text inside this area is examined. Only characters with fully contained bboxes are considered. The wrapper method `Page.searchFor()` correspondingly supports a `clip` parameter.
 3. Words **hyphenated** at the end of a line are now found.
 4. **Overlapping rectangles** in the same line are now automatically joined. We assume that such separations are an artifact created by multiple marked content groups, containing parts of the same search needle.
-

Example Quad versus Rect: when searching for needle “pymupdf”, then the corresponding entry will either be the blue rectangle, or, if *quads* was specified, the quad `Quad(ul, ur, ll, lr)`.

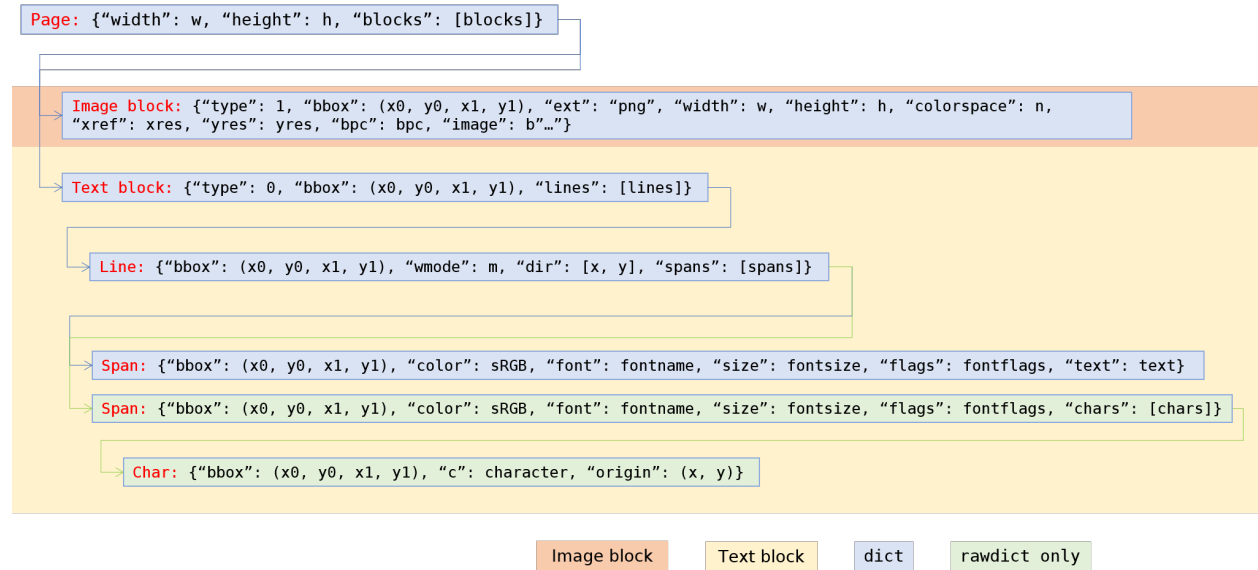
**rect**

The rectangle associated with the text page. This either equals the rectangle of the creating page or the `clip` parameter of `Page.getTextPage()` and text extraction / searching methods.

Note: The output of text searching and most text extractions **is restricted to this rectangle**. (X)HTML and XML output will however always extract the full page.

6.18.1 Dictionary Structure of `extractDICT()` and `extractRAW_DICT()`

Visual overview: the `TextPage` dictionary structure



6.18.1.1 Page Dictionary

Key	Value
width	page width in pixels (<i>float</i>)
height	page height in pixels (<i>float</i>)
blocks	<i>list</i> of block dictionaries

6.18.1.2 Block Dictionaries

Blocks come in two different formats: **image blocks** and **text blocks**.

(*Changed in v1.18.0*) – new dict key *number*, the block number.

Image block:

Key	Value
type	1 = image (<i>int</i>)
bbox	block / image rectangle, formatted as <i>tuple(fitz.Rect)</i>
number	block number (<i>int</i>) (0-based)
ext	image type (<i>str</i>), as file extension, see below
width	original image width (<i>int</i>)
height	original image height (<i>int</i>)
colorspace	colorspace.n (<i>int</i>)
xres	resolution in x-direction (<i>int</i>)
yres	resolution in y-direction (<i>int</i>)
bpc	bits per component (<i>int</i>)
image	image content (<i>bytes</i> or <i>bytearray</i>)

Possible values of key “ext” are “bmp”, “gif”, “jpeg”, “jpx” (JPEG 2000), “jxr” (JPEG XR), “png”, “pnm”, and “tiff”.

Note:

1. In some error situations, all of the above values may be zero or empty. So, please be prepared to digest items like:

```
{"type": 1, "bbox": (0.0, 0.0, 0.0, 0.0), ..., "image": b"..."}
```

2. *TextPage* and corresponding method *Page.getText()* are **available for all document types**. Only for PDF documents, methods *Document.getPageImageList()* / *Page.getImageList()* offer some overlapping functionality as far as image lists are concerned. But both lists **may or may not** contain the same items. Any differences are most probably caused by one of the following:
 - “Inline” images (see page 352 of the *Adobe PDF References*) of a PDF page are contained in a textpage, but **not in** *Page.getImageList()*.
 - Image blocks in a textpage are generated for **every** image location – whether or not there are any duplicates. This is in contrast to *Page.getImageList()*, which will contain each image only once.
 - Images mentioned in the page’s *object* definition will **always** appear in *Page.getImageList()*¹. But it may happen, that there is no “display” command in the page’s *contents* (erroneously or on purpose). In this case the image will **not appear** in the textpage.
-

Text block:

Key	Value
type	0 = text (<i>int</i>)
bbox	block rectangle, formatted as <i>tuple(fitz.Rect)</i>
number	block number (<i>int</i>) (0-based)
lines	<i>list</i> of text line dictionaries

6.18.1.3 Line Dictionary

Key	Value
bbox	line rectangle, formatted as <i>tuple(fitz.Rect)</i>
wmode	writing mode (<i>int</i>): 0 = horizontal, 1 = vertical
dir	writing direction (<i>list of floats</i>): $[x, y]$
spans	<i>list</i> of span dictionaries

The value of key “dir” is a **unit vector** and should be interpreted as follows:

- *x*: positive = “left-right”, negative = “right-left”, 0 = neither
- *y*: positive = “top-bottom”, negative = “bottom-top”, 0 = neither

The values indicate the “relative writing speed” in each direction, such that $x^2 + y^2 = 1$. In other words $dir = [\cos(\beta), \sin(\beta)]$, where β is the writing angle relative to the *x*-axis.

¹ Image specifications for a PDF page are done in a page’s (sub-) *dictionary*, called “/Resources”. Resource dictionaries can be **inherited** from the page’s parent object (usually the *catalog*). The PDF creator may e.g. define one /Resources on file level, naming all images and all fonts ever used by any page. In this case, *Page.getImageList()* and *Page.getFontList()* will always return the same lists for all pages.

6.18.1.4 Span Dictionary

Spans contain the actual text. A line contains **more than one span only**, if it contains text with different font properties. (Changed in version 1.14.17) Spans now also have a *bbox* key (again). (Changed in version 1.17.6) Spans now also have an *origin* key.

Key	Value
<i>bbox</i>	span rectangle, formatted as <i>tuple(fitz.Rect)</i>
<i>origin</i>	<i>tuple</i> coordinates of the first character's bottom left point
<i>font</i>	font name (<i>str</i>)
<i>ascender</i>	ascender of the font (<i>float</i>)
<i>descender</i>	descender of the font (<i>float</i>)
<i>size</i>	font size (<i>float</i>)
<i>flags</i>	font characteristics (<i>int</i>)
<i>color</i>	text color in sRGB format (<i>int</i>)
<i>text</i>	(only for <i>extractDICT()</i>) text (<i>str</i>)
<i>chars</i>	(only for <i>extractRAWDICT()</i>) <i>list</i> of character dictionaries

(New in version 1.16.0): “*color*” is the text color encoded in sRGB (int) format, e.g. 0xFF0000 for red. There are functions for converting this integer back to formats (r, g, b) (PDF with float values from 0 to 1) *sRGB_to_pdf()*, or (R, G, B), *sRGB_to_rgb()* (with integer values from 0 to 255).

(New in v1.18.5): “*ascender*” and “*descender*” are font properties, provided relative to fontsize 1. Note that descender is a negative value. The following picture shows the relationship to other values and properties.



These numbers may be used to compute the minimum height of a character (or span) – as opposed to the standard height provided in the “*bbox*” values (which actually represents the **line height**). The following code recalculates the span *bbox* to have a height of **fontsize** exactly fitting the text inside:

```
>>> a = span["ascender"]
>>> d = span["descender"]
>>> r = fitz.Rect(span["bbox"])
>>> o = fitz.Point(span["origin"]) # its y-value is the baseline
>>> r.y1 = o.y - span["size"] * d / (a - d)
>>> r.y0 = r.y1 - span["size"]
>>> # r now is a rectangle of height 'fontsize'
```

The following shows the original span rectangle in red and the rectangle with re-computed height in blue.



“*flags*” is an integer, interpreted as a bit field like this:

- bit 0: superscripted (2^0)
- bit 1: italic (2^1)
- bit 2: serifed (2^2)
- bit 3: monospaced (2^3)
- bit 4: bold (2^4)

Test these characteristics like so:

```
>>> if flags & 2**1: print ("italic")
>>> # etc.
```

6.18.1.5 Character Dictionary for `extractRAW_DICT()`

We are currently providing the bbox in *rect_like* format. In a future version, we might change that to *quad_like*.



This image shows the relationship between items in the following table:

Key	Value
origin	<i>tuple</i> coordinates of the character’s bottom left point
bbox	character rectangle, formatted as <i>tuple(fitz.Rect)</i>
c	the character (unicode)

6.19 TextWriter

(New in v1.16.18)

This class represents a MuPDF *text* object. The basic idea is to **decouple (1) text preparation, and (2) text output** to PDF pages.

During **preparation**, a text writer stores any number of text pieces (“spans”) together with their positions and individual font information. The **output** of the writer’s prepared content may happen multiple times to any PDF page with a compatible page size.

A text writer is an elegant alternative to methods `Page.insertText()` and friends:

- **Improved text positioning:** Choose any point where insertion of text should start. Storing text returns the “cursor position” after the *last character* of the span.

- **Free font choice:** Each text span has its own font and fontsize. This lets you easily switch when composing a larger text.
- **Automatic fallback fonts:** If a character is not supported by the chosen font, alternative fonts are automatically searched. This significantly reduces the risk of seeing unprintable symbols in the output (“TOFUs” – looking like a small rectangle). PyMuPDF now also comes with the **universal font “Droid Sans Fallback Regular”**, which supports **all Latin** characters (including Cyrillic and Greek), and **all CJK** characters (Chinese, Japanese, Korean).
- **Cyrillic and Greek Support:** The *PDF Base 14 Fonts* have integrated support of Cyrillic and Greek characters **without specifying encoding**. Your text may be a mixture of Latin, Greek and Cyrillic.
- **Transparency support:** Parameter *opacity* is supported. This offers a handy way to create watermark-style text.
- **Justified text:** Supported for any font – not just simple fonts as in *Page.insertTextbox()*.
- **Reusability:** A TextWriter object exists independent from PDF pages. It can be written multiple times, either to the same or to other pages, in the same or in different PDFs, choosing different colors or transparency.

Using this object entails three steps:

1. When **created**, a TextWriter requires a fixed **page rectangle** in relation to which it calculates text positions. A text writer can write to a page with the same size only.
2. Store text in the TextWriter using methods *TextWriter.append()*, *TextWriter.appendv()* and *TextWriter.fillTextbox()* as often as desired.
3. Output the TextWriter object on some PDF page.

Note:

- Starting with version 1.17.0, TextWriters **do support** text rotation via the *morph* parameter of *TextWriter.writeText()*.
 - There also exists *Page.writeText()* which combines one or more TextWriters and jointly writes them to a given rectangle and with a given rotation angle – much like *Page.showPDFpage()*.
-

Method / Attribute	Short Description
<i>append()</i>	Add text in horizontal write mode
<i>appendv()</i>	Add text in vertical write mode
<i>fillTextbox()</i>	Fill rectangle (horizontal write mode)
<i>writeText()</i>	Output TextWriter to a PDF page
<i>color</i>	Text color (can be changed)
<i>lastPoint</i>	Last written character ends here
<i>opacity</i>	Text opacity (can be changed)
<i>rect</i>	Page rectangle used by this TextWriter
<i>textRect</i>	Area occupied so far

Class API

```
class TextWriter
```

```
    __init__(self, rect, opacity=1, color=None)
```

Parameters

- **rect** (*rect-like*) – rectangle internally used for text positioning computations.

- **opacity** (*float*) – sets the transparency for the text to store here. Values outside the interval $[0, 1)$ will be ignored. A value of e.g. 0.5 means 50% transparency.
- **color** (*float, sequ*) – the color of the text. All colors are specified as floats $0 \leq color \leq 1$. A single float represents some gray level, a sequence implies the colorspace via its length.

append (*pos, text, font=None, fontsize=11, language=None*)

Add some new text in horizontal, left-to-right writing.

Parameters

- **pos** (*point_like*) – start position of the text, the bottom left point of the first character.
- **text** (*str*) – a string (Python 2: unicode is mandatory!) of arbitrary length. It will be written starting at position “pos”.
- **font** – a *Font*. If omitted, `fitz.Font("helv")` will be used.
- **fontsize** (*float*) – the fontsize, a positive number, default 11.
- **language** (*str*) – the language to use, e.g. “en” for English. Meaningful values should be compliant with the ISO 639 standards 1, 2, 3 or 5. Reserved for future use: currently has no effect as far as we know.

Returns *textRect* and *lastPoint*. (*Changed in v1.18.0:*) Raises an exception for an unsupported font – checked via *Font.isWritable*.

appendv (*pos, text, font=None, fontsize=11, language=None*)

Add some new text in vertical, top-to-bottom writing.

Parameters

- **pos** (*point_like*) – start position of the text, the bottom left point of the first character.
- **text** (*str*) – a string (Python 2: unicode is mandatory!) of arbitrary length. It will be written starting at position “pos”.
- **font** – a *Font*. If omitted, `fitz.Font("helv")` will be used.
- **fontsize** (*float*) – the fontsize, a positive number, default 11.
- **language** (*str*) – the language to use, e.g. “en” for English. Meaningful values should be compliant with the ISO 639 standards 1, 2, 3 or 5. Reserved for future use: currently has no effect as far as we know.

Returns *textRect* and *lastPoint*. (*Changed in v1.18.0:*) Raises an exception for an unsupported font – checked via *Font.isWritable*.

fillTextbox (*rect, text, pos=None, font=None, fontsize=11, align=0, warn=True*)

Fill a given rectangle with text in horizontal, left-to-right manner. This is a convenience method to use as an alternative to *append()*.

Parameters

- **rect** (*rect_like*) – the area to fill. No part of the text will appear outside of this.
- **text** (*str, sequ*) – the text. Can be specified as a (UTF-8) string or a list / tuple of strings. A string will first be converted to a list using *splitlines()*. Every list item will begin on a new line (forced line breaks).
- **pos** (*point_like*) – (*new in v1.17.3*) start storing at this point. Default is a point near rectangle top-left.

- **font** – the *Font*, default *fitz.Font("helv")*.
- **fontsize** (*float*) – the fontsize.
- **align** (*int*) – text alignment. Use one of `TEXT_ALIGN_LEFT`, `TEXT_ALIGN_CENTER`, `TEXT_ALIGN_RIGHT` or `TEXT_ALIGN_JUSTIFY`.
- **warn** (*bool*) – warn on text overflow (default), or raise an exception. In any case, text not fitting will not be written.

Note: Use these methods as often as is required – there is no technical limit (except memory constraints of your system). You can also mix appends and text boxes and have multiple of both. Text positioning is controlled by the insertion point. There is no need to adhere to any order. (*Changed in v1.18.0:*) Raises an exception for an unsupported font – checked via *Font.isWritable*.

writeText (*page, opacity=None, color=None, morph=None, overlay=True, oc=0*)

Write the TextWriter text to a page.

Parameters

- **page** – write to this *Page*.
- **opacity** (*float*) – override the value of the TextWriter for this output.
- **color** (*sequ*) – override the value of the TextWriter for this output.
- **morph** (*sequ*) – modify the text appearance by applying a matrix to it. If provided, this must be a sequence (*fixpoint, matrix*) with a point-like *fixpoint* and a matrix-like *matrix*. A typical example is rotating the text around *fixpoint*.
- **overlay** (*bool*) – put in foreground (default) or background.
- **oc** (*int*) – (*new in v1.18.4*) the *xref* of an *OCG* or *OCMD*.

textRect

Return type *Rect*

The area currently occupied.

lastPoint

Return type *Point*

The “cursor position” – a *Point* – after the last written character (its bottom-right).

opacity

The text opacity (modifyable).

color

The text color (modifyable).

rect

The page rectangle for which this TextWriter was created. Must not be modified.

To see some demo scripts dealing with TextWriter, have a look at [this](#) repository.

Note:

1. Opacity and color apply to **all the text** in this object.

2. If you need different colors / transparency, you must create a separate `TextWriter`. Whenever you determine the color should change, simply append the text to the respective `TextWriter` using the previously returned `lastPoint` as position for the new text span.
 3. Appending items or text boxes can occur in arbitrary order: only the position parameter controls where text appears.
 4. Font and fontsize can freely vary within the same `TextWriter`. This can be used to let text with different properties appear on the same displayed line: just specify *pos* accordingly, and e.g. set it to `lastPoint` of the previously added item.
 5. You can use the *pos* argument of `TextWriter.fillTextbox()` to indent the first line, so its text may continue any preceeding one in a continuous manner.
 6. MuPDF does not support all fonts with this feature, e.g. no Type3 fonts. Starting with v1.18.0 this can be checked via the font attribute `Font.isWritable`.
-

6.20 Tools

This class is a collection of utility methods and attributes, mainly around memory management. To simplify and speed up its use, it is automatically instantiated under the name `TOOLS` when PyMuPDF is imported.

Method / Attribute	Description
<code>Tools.gen_id()</code>	generate a unique identifier
<code>Tools.image_profile()</code>	report basic image properties
<code>Tools.store_shrink()</code>	shrink the storables cache ¹
<code>Tools.mupdf_warnings()</code>	return the accumulated MuPDF warnings
<code>Tools.mupdf_display_errors()</code>	return the accumulated MuPDF warnings
<code>Tools.reset_mupdf_warnings()</code>	empty MuPDF messages on STDOUT
<code>Tools.set_aa_level()</code>	set the anti-aliasing values
<code>Tools.set_small_glyph_heights()</code>	search and extract small bbox heights
<code>Tools.show_aa_level()</code>	return the anti-aliasing values
<code>Tools.fitz_config</code>	configuration settings of PyMuPDF
<code>Tools.store_maxsize</code>	maximum storables cache size
<code>Tools.store_size</code>	current storables cache size

Class API

`class Tools`

`gen_id()`

A convenience method returning a unique positive integer which will increase by 1 on every invocation. Example usages include creating unique keys in databases - its creation should be faster than using timestamps by an order of magnitude.

Note: MuPDF has dropped support for this in v1.14.0, so we have re-implemented a similar function with the following differences:

¹ This memory area is internally used by MuPDF, and it serves as a cache for objects that have already been read and interpreted, thus improving performance. The most bulky object types are images and also fonts. When an application starts up the MuPDF library (in our case this happens as part of `import fitz`), it must specify a maximum size for this area. PyMuPDF's uses the default value (256 MB) to limit memory consumption. Use the methods here to control or investigate store usage. For example: even after a document has been closed and all related objects have been deleted, the store usage may still not drop down to zero. So you might want to enforce that before opening another document.

- It is not part of MuPDF’s global context and not threadsafe (not an issue because we do not support threads in PyMuPDF anyway).
- It is implemented as *int*. This means that the maximum number is *sys.maxsize*. Should this number ever be exceeded, the counter starts over again at 1.

Return type *int*

Returns a unique positive integer.

set_small_glyph_heights (*on=None*)
(New in v1.18. – experimental)

Set or inquire reduced bbox heights in text extract and text search methods.

Parameters *on* (*bool*) – if omitted or *None*, the current setting is returned. For other values the *bool()* function is applied to set a global variable. If *True*, *Page.searchFor()* and *Page.getText()* methods return character, span, line or block bboxes that have a height of *font size*. If *False* (the standard setting when PyMuPDF is imported), bbox height will normally equal *line height*.

Return type *bool*

Returns *True* or *False*.

image_profile (*stream*)

(New in v1.16.17) Show important properties of an image provided as a memory area. Its main purpose is to avoid using other Python packages just to determine basic properties.

Parameters *stream* (*bytes, bytearray*) – the image data.

Return type *dict*

Returns a dictionary with the keys “width”, “height”, “xres”, “yres”, “colorspace” (the *colorspace.n* value, number of colorants), “cs-name” (the *colorspace.name* value), “bpc”, “ext” (image type as file extension). The values for these keys are the same as returned by *Document.extractImage()*. Please also have a look at *resolution*.

Note:

- For some “exotic” images (FAX encodings, RAW formats and the like), this method will not work and return *None*. You can however still work with such images in PyMuPDF, e.g. by using *Document.extractImage()* or create pixmaps via *Pixmap(doc, xref)*. These methods will automatically convert exotic images to the PNG format before returning results.
- Some examples:

```
In [1]: import fitz
In [2]: stream = open(<image.file>, "rb").read()
In [3]: fitz.TOOLS.image_profile(stream)
Out[3]:
{'width': 439,
'height': 501,
'xres': 96,
'yres': 96,
'colorspace': 3,
'bpc': 8,
'ext': 'jpeg',
'cs-name': 'DeviceRGB'}
```

(continues on next page)

(continued from previous page)

```

In [4]: doc=fitz.open(<input.pdf>)
In [5]: stream = doc.xrefStreamRaw(5) # no decompression!
In [6]: fitz.TOOLS.image_profile(stream)
Out[6]:
{'width': 816,
'height': 1056,
'xres': 96,
'yres': 96,
'colorspace': 1,
'bpc': 8,
'ext': 'jpeg',
'cs-name': 'DeviceGray'}

```

store_shrink (*percent*)

Reduce the storables cache by a percentage of its current size.

Parameters **percent** (*int*) – the percentage of current size to free. If 100+ the store will be emptied, if zero, nothing will happen. MuPDF’s caching strategy is “least recently used”, so low-usage elements get deleted first.

Return type *int*

Returns the new current store size. Depending on the situation, the size reduction may be larger than the requested percentage.

show_aa_level ()

(*New in version 1.16.14*) Return the current anti-aliasing values. These values control the rendering quality of graphics and text elements.

Return type *dict*

Returns A dictionary with the following initial content: {'graphics': 8, 'text': 8, 'graphics_min_line_width': 0.0}.

set_aa_level (*level*)

(*New in version 1.16.14*) Set the new number of bits to use for anti-aliasing. The same value is taken currently for graphics and text rendering. This might change in a future MuPDF release.

Parameters **level** (*int*) – an integer ranging between 0 and 8. Value outside this range will be silently changed to valid values. The value will remain in effect throughout the current session or until changed again.

reset_mupdf_warnings ()

(*New in version 1.16.0*)

Empty MuPDF warnings message buffer.

mupdf_display_errors (*value=None*)

(*New in version 1.16.8*)

Show or set whether MuPDF errors should be displayed.

Parameters **value** (*bool*) – if not a bool, the current setting is returned. If true, MuPDF errors will be shown on *sys.stderr*, otherwise suppressed. In any case, messages continue to be stored in the warnings store. Upon import of PyMuPDF this value is *True*.

Returns *True* or *False*

mupdf_warnings (*reset=True*)

(*New in version 1.16.0*)

Return all stored MuPDF messages as a string with interspersed line-breaks.

Parameters `reset` (*bool*) – (*new in version 1.16.7*) whether to automatically empty the store.

`fitz_config`

A dictionary containing the actual values used for configuring PyMuPDF and MuPDF. Also refer to the installation chapter. This is an overview of the keys, each of which describes the status of a support aspect.

Key	Support included for ...
plotter-g	Gray colorspace rendering
plotter-rgb	RGB colorspace rendering
plotter-cmyk	CMYK colorspace rendering
plotter-n	overprint rendering
pdf	PDF documents
xps	XPS documents
svg	SVG documents
cbz	CBZ documents
img	IMG documents
html	HTML documents
epub	EPUB documents
jpx	JPEG2000 images
js	JavaScript
tofu	all TOFU fonts
tofu-cjk	CJK font subset (China, Japan, Korea)
tofu-cjk-ext	CJK font extensions
tofu-cjk-lang	CJK font language extensions
tofu-emoji	TOFU emoji fonts
tofu-historic	TOFU historic fonts
tofu-symbol	TOFU symbol fonts
tofu-sil	TOFU SIL fonts
icc	ICC profiles
py-memory	using Python memory management ²
base14	Base-14 fonts (should always be true)

For an explanation of the term “TOFU” see [this Wikipedia article](#).

```
In [1]: import fitz
In [2]: TOOLS.fitz_config
Out[2]:
{'plotter-g': True,
 'plotter-rgb': True,
 'plotter-cmyk': True,
 'plotter-n': True,
 'pdf': True,
 'xps': True,
 'svg': True,
 'cbz': True,
 'img': True,
```

(continues on next page)

² Optionally, all dynamic management of memory can be done using Python C-level calls. MuPDF offers a hook to insert user-preferred memory managers. We are using option this for Python version 3 since PyMuPDF v1.13.19. At the same time, all memory allocation in PyMuPDF itself is also routed to Python (i.e. no more direct `malloc()` calls in the code). We have seen improved memory usage and slightly reduced runtimes with this option set. If you want to change this, you can set `#define JM_MEMORY 0` (uses standard C malloc, or 1 for Python allocation) in file `fitz.i` and then generate PyMuPDF.

(continued from previous page)

```
'html': True,
'epub': True,
'jpx': True,
'js': True,
'tofu': False,
'tofu-cjk': True,
'tofu-cjk-ext': False,
'tofu-cjk-lang': False,
'tofu-emoji': False,
'tofu-historic': False,
'tofu-symbol': False,
'tofu-sil': False,
'icc': True,
'py-memory': True, # (False if Python 2)
'base14': True}
```

Return type dict**store_maxsize**

Maximum storables cache size in bytes. PyMuPDF is generated with a value of 268'435'456 (256 MB, the default value), which you should therefore always see here. If this value is zero, then an “unlimited” growth is permitted.

Return type int**store_size**

Current storables cache size in bytes. This value may change (and will usually increase) with every use of a PyMuPDF function. It will (automatically) decrease only when `Tools.store_maxsize` is going to be exceeded: in this case, MuPDF will evict low-usage objects until the value is again in range.

Return type int

6.20.1 Example Session

::

```
>>> import fitz
# print the maximum and current cache sizes
>>> fitz.TOOLS.store_maxsize
268435456
>>> fitz.TOOLS.store_size
0
>>> doc = fitz.open("demol.pdf")
# pixmap creation puts lots of object in cache (text, images, fonts),
# apart from the pixmap itself
>>> pix = doc[0].getPixmap(alpha=False)
>>> fitz.TOOLS.store_size
454519
# release (at least) 50% of the storage
>>> fitz.TOOLS.store_shrink(50)
13471
>>> fitz.TOOLS.store_size
13471
# get a few unique numbers
>>> fitz.TOOLS.gen_id()
```

(continues on next page)

(continued from previous page)

```

1
>>> fitz.TOOLS.gen_id()
2
>>> fitz.TOOLS.gen_id()
3
# close document and see how much cache is still in use
>>> doc.close()
>>> fitz.TOOLS.store_size
0
>>>

```

6.21 Widget

This class represents a PDF Form field, also called a “widget”. Throughout this documentation, we are using these terms synonymously. Fields technically are a special case of PDF annotations, which allow users with limited permissions to enter information in a PDF. This is primarily used for filling out forms.

Like annotations, widgets live on PDF pages. Similar to annotations, the first widget on a page is accessible via `Page.firstWidget` and subsequent widgets can be accessed via the `Widget.next` property.

(*Changed in version 1.16.0*) MuPDF no longer treats widgets as a subset of general annotations. Consequently, `Page.firstAnnot` and `Annot.next()` will deliver **non-widget annotations exclusively**, and be `None` if only form fields exist on a page. Vice versa, `Page.firstWidget` and `Widget.next()` will only show widgets. This design decision is purely internal to MuPDF; technically, links, annotations and fields have a lot in common and also continue to share the better part of their code within (Py-) MuPDF.

Class API

`class Widget`

`update()`

After any changes to a widget, this method **must be used** to store them in the PDF¹.

`reset()`

Reset the field’s value to its default – if defined – or remove it. Do not forget to issue `update()` afterwards.

`next`

Point to the next form field on the page. The last widget returns `None`.

`border_color`

A list of up to 4 floats defining the field’s border color. Default value is `None` which causes border style and border width to be ignored.

`border_style`

A string defining the line style of the field’s border. See `Annot.border`. Default is “s” (“Solid”) – a continuous line. Only the first character (upper or lower case) will be regarded when creating a widget.

`border_width`

A float defining the width of the border line. Default is 1.

`border_dashes`

A list/tuple of integers defining the dash properties of the border line. This is only meaningful if `border_style == “D”` and `border_color` is provided.

¹ If you intend to re-access a new or updated field (e.g. for making a pixmap), make sure to reload the page first. Either close and re-open the document, or load another page first, or simply do `page = doc.reload_page(page)`.

choice_values

Python sequence of strings defining the valid choices of list boxes and combo boxes. For these widget types, this property is mandatory and must contain at least two items. Ignored for other types.

field_name

A mandatory string defining the field's name. No checking for duplicates takes place.

field_label

An optional string containing an "alternate" field name. Typically used for any notes, help on field usage, etc. Default is the field name.

field_value

The value of the field.

field_flags

An integer defining a large amount of properties of a field. Be careful when changing this attribute as this may change the field type.

field_type

A mandatory integer defining the field type. This is a value in the range of 0 to 6. It cannot be changed when updating the widget.

field_type_string

A string describing (and derived from) the field type.

fill_color

A list of up to 4 floats defining the field's background color.

button_caption

The caption string of a button-type field.

is_signed

A bool indicating the signing status of a signature field, else *None*.

rect

The rectangle containing the field.

text_color

A list of **1, 3 or 4 floats** defining the text color. Default value is black $([0, 0, 0])$.

text_font

A string defining the font to be used. Default and replacement for invalid values is "*Helv*". For valid font reference names see the table below.

text_fontsize

A float defining the text fontsize. Default value is zero, which causes PDF viewer software to dynamically choose a size suitable for the annotation's rectangle and text amount.

text_maxlen

An integer defining the maximum number of text characters. PDF viewers will (should) not accept a longer text.

text_type

An integer defining acceptable text types (e.g. numeric, date, time, etc.). For reference only for the time being – will be ignored when creating or updating widgets.

xref

The PDF *xref* of the widget.

script

(*New in version 1.16.12*) JavaScript text (unicode) for an action associated with the widget, or *None*. This is the only script action supported for **button type** widgets.

script_stroke

(New in version 1.16.12) JavaScript text (unicode) to be performed when the user types a key-stroke into a text field or combo box or modifies the selection in a scrollable list box. This action can check the keystroke for validity and reject or modify it. *None* if not present.

script_format

(New in version 1.16.12) JavaScript text (unicode) to be performed before the field is formatted to display its current value. This action can modify the field's value before formatting. *None* if not present.

script_change

(New in version 1.16.12) JavaScript text (unicode) to be performed when the field's value is changed. This action can check the new value for validity. *None* if not present.

script_calc

(New in version 1.16.12) JavaScript text (unicode) to be performed to recalculate the value of this field when that of another field changes. *None* if not present.

Note:

1. For **adding** or **changing** one of the above scripts, just put the appropriate JavaScript source code in the widget attribute. To **remove** a script, set the respective attribute to *None*.
2. Button fields only support *script*. Other script entries will automatically be set to *None*.

6.21.1 Standard Fonts for Widgets

Widgets use their own resources object */DR*. A widget resources object must at least contain a */Font* object. Widget fonts are independent from page fonts. We currently support the 14 PDF base fonts using the following fixed reference names, or any name of an already existing field font. When specifying a text font for new or changed widgets, **either** choose one in the first table column (upper and lower case supported), **or** one of the already existing form fonts. In the latter case, spelling must exactly match.

To find out already existing field fonts, inspect the list *Document.FormFonts*.

Reference	Base14 Fontname
CoBI	Courier-BoldOblique
CoBo	Courier-Bold
CoIt	Courier-Oblique
Cour	Courier
HeBI	Helvetica-BoldOblique
HeBo	Helvetica-Bold
HeIt	Helvetica-Oblique
Helv	Helvetica (default)
Symb	Symbol
TiBI	Times-BoldItalic
TiBo	Times-Bold
TiIt	Times-Italic
TiRo	Times-Roman
ZaDb	ZapfDingbats

You are generally free to use any font for every widget. However, we recommend using *ZaDb* (“ZapfDingbats”) and fontsize 0 for check boxes: typical viewers will put a correctly sized tickmark in the field's rectangle, when it is clicked.

6.21.2 Supported Widget Types

PyMuPDF supports the creation and update of many, but not all widget types.

- text (PDF_WIDGET_TYPE_TEXT)
- push button (PDF_WIDGET_TYPE_BUTTON)
- check box (PDF_WIDGET_TYPE_CHECKBOX)
- combo box (PDF_WIDGET_TYPE_COMBOBOX)
- list box (PDF_WIDGET_TYPE_LISTBOX)
- radio button (PDF_WIDGET_TYPE_RADIOBUTTON): PyMuPDF does not currently support groups of (inter-connected) buttons, where setting one automatically unsets the other buttons in the group. The widget object also does not reflect the presence of a button group. Setting or unsetting happens via values `True` and `False` and will always work without affecting other radio buttons.
- signature (PDF_WIDGET_TYPE_SIGNATURE) **read only**.

Operator Algebra for Geometry Objects

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

They all are special cases of Python sequences, see *Using Python Sequences as Arguments in PyMuPDF* for more background.

We have defined operators for these classes that allow dealing with them (almost) like ordinary numbers in terms of addition, subtraction, multiplication, division, and some others.

This chapter is a synopsis of what is possible.

7.1 General Remarks

1. Operators can be either **binary** (i.e. involving two objects) or **unary**.
2. The resulting type of **binary** operations is either a **new object of the left operand’s class** or a bool.
3. The result of **unary** operations is either a **new object** of the same class, a bool or a float.
4. The binary operators $+$, $-$, $*$, $/$ are defined for all classes. They *roughly* do what you would expect – **except, that the second operand ...**
 - may always be a number which then performs the operation on every component of the first one,
 - may always be a numeric sequence of the same length (2, 4 or 6) – we call such sequences *point_like*, *rect_like* or *matrix_like*, respectively.
5. Rectangles support additional binary operations: **intersection** (operator “&”), **union** (operator “|”) and **containment** checking.
6. Binary operators fully support in-place operations, so expressions like “ $a /= b$ ” are valid if b is numeric or “a_like”.

7.2 Unary Operations

Oper.	Result
bool(OBJ)	is false exactly if all components of OBJ are zero
abs(OBJ)	the rectangle area – equal to norm(OBJ) for the other types
norm(OBJ)	square root of the component squares (Euclidean norm)
+OBJ	new copy of OBJ
-OBJ	new copy of OBJ with negated components
~m	inverse of matrix “m”, or the null matrix if not invertible

7.3 Binary Operations

For every geometry object “a” and every number “b”, the operations “a ° b” and “a °= b” are always defined for the operators +, -, *, /. The respective operation is simply executed for each component of “a”. If the **second operand is not a number**, then the following is defined:

Oper.	Result
a+b, a-b	component-wise execution, “b” must be “a-like”.
a*m, a/m	“a” can be a point, rectangle or matrix, but “m” must be <i>matrix_like</i> . “a/m” is treated as “a*~m” (see note below for non-invertible matrices). If “a” is a point or a rectangle , then “a.transform(m)” is executed. If “a” is a matrix, then matrix concatenation takes place.
a&b	intersection rectangle : “a” must be a rectangle and “b” <i>rect_like</i> . Delivers the largest rectangle contained in both operands.
alb	union rectangle : “a” must be a rectangle, and “b” may be <i>point_like</i> or <i>rect_like</i> . Delivers the smallest rectangle containing both operands.
b in a	if “b” is a number, then “b in tuple(a)” is returned. If “b” is <i>point_like</i> or <i>rect_like</i> , then “a” must be a rectangle, and “a.contains(b)” is returned.
a == b	<i>True</i> if <i>bool(a-b)</i> is <i>False</i> (“b” may be “a-like”).

Note: Please note an important difference to usual arithmetics:

Matrix multiplication is **not commutative**, i.e. in general we have $m*n \neq n*m$ for two matrices. Also, there are non-zero matrices which have no inverse, for example $m = \text{Matrix}(1, 0, 1, 0, 1, 0)$. If you try to divide by any of these you will receive a *ZeroDivisionError* exception using operator “/”, e.g. for *fitz.Identity / m*. But if you formulate *fitz.Identity * ~m*, the result will be *fitz.Matrix()* (the null matrix).

Admittedly, this represents an inconsistency, and we are considering to remove it. For the time being, you can choose to avoid an exception and check whether ~m is the null matrix, or accept a potential *ZeroDivisionError* by using *fitz.Identity / m*.

7.4 Some Examples

7.4.1 Manipulation with numbers

For the usual arithmetic operations, numbers are always allowed as second operand. In addition, you can formulate “*x in OBJ*”, where *x* is a number. It is implemented as “*x in tuple(OBJ)*”:

```
>>> fitz.Rect(1, 2, 3, 4) + 5
fitz.Rect(6.0, 7.0, 8.0, 9.0)
>>> 3 in fitz.Rect(1, 2, 3, 4)
True
>>>
```

The following will create the upper left quarter of a document page rectangle:

```
>>> page.rect
Rect(0.0, 0.0, 595.0, 842.0)
>>> page.rect / 2
Rect(0.0, 0.0, 297.5, 421.0)
>>>
```

The following will deliver the **middle point of a line** connecting two points **p1** and **p2**:

```
>>> p1 = fitz.Point(1, 2)
>>> p2 = fitz.Point(4711, 3141)
>>> mp = p1 + (p2 - p1) / 2
>>> mp
Point(2356.0, 1571.5)
>>>
```

7.4.2 Manipulation with “like” Objects

The second operand of a binary operation can always be “like” the left operand. “Like” in this context means “a sequence of numbers of the same length”. With the above examples:

```
>>> p1 + p2
Point(4712.0, 3143.0)
>>> p1 + (4711, 3141)
Point(4712.0, 3143.0)
>>> p1 += (4711, 3141)
>>> p1
Point(4712.0, 3143.0)
>>>
```

To shift a rectangle for 5 pixels to the right, do this:

```
>>> fitz.Rect(100, 100, 200, 200) + (5, 0, 5, 0) # add 5 to the x coordinates
Rect(105.0, 100.0, 205.0, 200.0)
>>>
```

Points, rectangles and matrices can be *transformed* with matrices. In PyMuPDF, we treat this like a “**multiplication**” (or resp. “**division**”), where the second operand may be “like” a matrix. Division in this context means “multiplication with the inverted matrix”:

```

>>> m = fitz.Matrix(1, 2, 3, 4, 5, 6)
>>> n = fitz.Matrix(6, 5, 4, 3, 2, 1)
>>> p = fitz.Point(1, 2)
>>> p * m
Point(12.0, 16.0)
>>> p * (1, 2, 3, 4, 5, 6)
Point(12.0, 16.0)
>>> p / m
Point(2.0, -2.0)
>>> p / (1, 2, 3, 4, 5, 6)
Point(2.0, -2.0)
>>>
>>> m * n # matrix multiplication
Matrix(14.0, 11.0, 34.0, 27.0, 56.0, 44.0)
>>> m / n # matrix division
Matrix(2.5, -3.5, 3.5, -4.5, 5.5, -7.5)
>>>
>>> m / m # result is equal to the Identity matrix
Matrix(1.0, 0.0, 0.0, 1.0, 0.0, 0.0)
>>>
>>> # look at this non-invertible matrix:
>>> m = fitz.Matrix(1, 0, 1, 0, 1, 0)
>>> ~m
Matrix(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
>>> # we try dividing by it in two ways:
>>> p = fitz.Point(1, 2)
>>> p * ~m # this delivers point (0, 0):
Point(0.0, 0.0)
>>> p / m # but this is an exception:
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    p / m
  File "... /site-packages/fitz/fitz.py", line 869, in __truediv__
    raise ZeroDivisionError("matrix not invertible")
ZeroDivisionError: matrix not invertible
>>>

```

As a specialty, rectangles support additional binary operations:

- **intersection** – the common area of rectangle-likes, operator “&”
- **inclusion** – enlarge to include a point-like or rect-like, operator “|”
- **containment** check – whether a point-like or rect-like is inside

Here is an example for creating the smallest rectangle enclosing given points:

```

>>> # first define some point-likes
>>> points = []
>>> for i in range(10):
    for j in range(10):
        points.append((i, j))
>>>
>>> # now create a rectangle containing all these 100 points
>>> # start with an empty rectangle
>>> r = fitz.Rect(points[0], points[0])
>>> for p in points[1:]: # and include remaining points one by one
    r |= p
>>> r # here is the to be expected result:

```

(continues on next page)

(continued from previous page)

```
Rect(0.0, 0.0, 9.0, 9.0)
>>> (4, 5) in r # this point-like lies inside the rectangle
True
>>> # and this rect-like is also inside
>>> (4, 4, 5, 5) in r
True
>>>
```

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.

8.1 Functions

The following are miscellaneous functions and attributes on a fairly low-level technical detail.

Some functions provide detail access to PDF structures. Others are stripped-down, high performance versions of other functions which provide more information.

Yet others are handy, general-purpose utilities.

Function	Short Description
<i>Annot.clean_contents()</i>	PDF only: clean the annot's <i>contents</i> object
<i>Annot.set_apn_matrix()</i>	PDF only: set the matrix of the appearance object
<i>Annot.set_apn_bbox()</i>	PDF only: set the bbox of the appearance object
<i>Annot.apn_matrix</i>	PDF only: the matrix of the appearance object
<i>Annot.apn_bbox</i>	PDF only: bbox of the appearance object
<i>ConversionHeader()</i>	return header string for <i>getText</i> methods
<i>ConversionTrailer()</i>	return trailer string for <i>getText</i> methods
<i>Document.del_xml_metadata()</i>	PDF only: remove XML metadata
<i>Document.set_xml_metadata()</i>	PDF only: remove XML metadata
<i>Document.delete_object()</i>	PDF only: delete an object
<i>Document.get_new_xref()</i>	PDF only: create and return a new <i>xref</i> entry
<i>Document._getOLRootNumber()</i>	PDF only: return / create <i>xref</i> of <i>/Outline</i>
<i>Document.pdf_catalog()</i>	PDF only: return the <i>xref</i> of the catalog
<i>Document.page_xref()</i>	PDF only: get <i>xref</i> of page object by page number
<i>Document.pdf_trailer()</i>	PDF only: return the PDF file trailer string
<i>Document.xml_metadata_xref()</i>	PDF only: return XML metadata <i>xref</i> number
<i>Document.xref_length()</i>	PDF only: return length of <i>xref</i> table

Continued on next page

Table 1 – continued from previous page

Function	Short Description
<code>Document.xref_object()</code>	PDF only: return object definition “source”
<code>Document._make_page_map()</code>	PDF only: create a fast-access array of page numbers
<code>Document.extractFont()</code>	PDF only: extract embedded font
<code>Document.extractImage()</code>	PDF only: extract embedded image
<code>Document.getCharWidths()</code>	PDF only: return a list of glyph widths of a font
<code>Document.isStream()</code>	PDF only: check whether an <i>xref</i> is a stream object
<code>Document.FontInfos</code>	PDF only: information on inserted fonts
<code>ImageProperties()</code>	return a dictionary of basic image properties
<code>getPDFnow()</code>	return the current timestamp in PDF format
<code>getPDFstr()</code>	return PDF-compatible string
<code>getTextlength()</code>	return string length for a given font & fontsize
<code>Page.clean_contents()</code>	PDF only: clean the page’s <i>contents</i> objects
<code>Page.get_contents()</code>	PDF only: return a list of content <i>xref</i> numbers
<code>Page.set_contents()</code>	PDF only: set page’s <i>contents</i> to some <i>xref</i>
<code>Page.getDisplayList()</code>	create the page’s display list
<code>Page.getTextBlocks()</code>	extract text blocks as a Python list
<code>Page.getTextWords()</code>	extract text words as a Python list
<code>Page.run()</code>	run a page through a device
<code>Page.read_contents()</code>	PDF only: get complete, concatenated /Contents source
<code>Page.wrap_contents()</code>	wrap contents with stacking commands
<code>Page.is_wrapped</code>	check whether contents wrapping is present
<code>planishLine()</code>	matrix to map a line to the x-axis
<code>PaperSize()</code>	return width, height for a known paper format
<code>PaperRect()</code>	return rectangle for a known paper format
<code>sRGB_to_pdf()</code>	return PDF RGB color tuple from a sRGB integer
<code>sRGB_to_rgb()</code>	return (R, G, B) color tuple from a sRGB integer
<code>glyph_name_to_unicode()</code>	return unicode from a glyph name
<code>unicode_to_glyph_name()</code>	return glyph name from a unicode
<code>make_table()</code>	split rectangle in sub-rectangles
<code>adobe_glyph_names()</code>	list of glyph names defined in Adobe Glyph List
<code>adobe_glyph_unicodes()</code>	list of unicodes defined in Adobe Glyph List
<code>paperSizes</code>	dictionary of pre-defined paper formats
<code>fitz_fontdescriptors</code>	dictionary of available supplement fonts

PaperSize(*s*)

Convenience function to return width and height of a known paper format code. These values are given in pixels for the standard resolution 72 pixels = 1 inch.

Currently defined formats include ‘A0’ through ‘A10’, ‘B0’ through ‘B10’, ‘C0’ through ‘C10’, ‘Card-4x6’, ‘Card-5x7’, ‘Commercial’, ‘Executive’, ‘Invoice’, ‘Ledger’, ‘Legal’, ‘Legal-13’, ‘Letter’, ‘Monarch’ and ‘Tabloid-Extra’, each in either portrait or landscape format.

A format name must be supplied as a string (case **in** sensitive), optionally suffixed with “-L” (landscape) or “-P” (portrait). No suffix defaults to portrait.

Parameters *s* (*str*) – any format name from above (upper or lower case), like “A4” or “letter-l”.

Return type tuple

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

PaperRect (*s*)

Convenience function to return a *Rect* for a known paper format.

Parameters *s* (*str*) – any format name supported by *PaperSize* ().

Return type *Rect*

Returns *fitz.Rect(0, 0, width, height)* with *width, height=fitz.PaperSize(s)*.

```
>>> import fitz
>>> fitz.PaperRect("letter-l")
fitz.Rect(0.0, 0.0, 792.0, 612.0)
>>>
```

sRGB_to_pdf (*srgb*)

New in v1.17.4

Convenience function returning a PDF color triple (red, green, blue) for a given sRGB color integer as it occurs in *Page.getText()* dictionaries “dict” and “rawdict”.

Parameters *srgb* (*int*) – an integer of format RRGGBB, where each color component is an integer in range(255).

Returns a tuple (red, green, blue) with float items in interval $0 \leq \text{item} \leq 1$ representing the same color.

glyph_name_to_unicode (*name*)

New in v1.18.0

Return the unicode number of a glyph name based on the **Adobe Glyph List**.

Parameters *name* (*str*) – the name of some glyph. The function is based on the [Adobe Glyph List](#).

Return type *int*

Returns the unicode. Invalid *name* entries return 0xffffd (65533).

Note: A similar functionality is provided by package [fontTools](#) in its *agl* sub-package.

unicode_to_glyph_name (*ch*)

New in v1.18.0

Return the glyph name of a unicode number, based on the **Adobe Glyph List**.

Parameters *ch* (*int*) – the unicode given by e.g. `ord("B")`. The function is based on the [Adobe Glyph List](#).

Return type *str*

Returns the glyph name. E.g. `fitz.unicode_to_glyph_name(ord("Ä"))` returns 'Adieresis'.

Note: A similar functionality is provided by package [fontTools](#): in its *agl* sub-package.

adobe_glyph_names ()

New in v1.18.0

Return a list of glyph names defined in the **Adobe Glyph List**.

Return type list

Returns list of strings.

Note: A similar functionality is provided by package `fontTools` in its *agl* sub-package.

adobe_glyph_unICODES ()

New in v1.18.0

Return a list of unicodes for there exists a glyph name in the **Adobe Glyph List**.

Return type list

Returns list of integers.

Note: A similar functionality is provided by package `fontTools` in its *agl* sub-package.

sRGB_to_rgb (*srgb*)

New in v1.17.4

Convenience function returning a color (red, green, blue) for a given *sRGB* color integer .

Parameters **srgb** (*int*) – an integer of format RRGGBB, where each color component is an integer in range(255).

Returns a tuple (red, green, blue) with integer items in intervall $0 \leq \text{item} \leq 255$ representing the same color.

make_table (*rect*, *cols=1*, *rows=1*)

New in v1.17.4

Convenience function to split a rectangle into sub-rectangles. Returns a list of *rows* lists, each containing *cols* *Rect* items. Each sub-rectangle can then be addressed by its row and column index.

Parameters

- **rect** (*rect_like*) – the rectangle to split.
- **cols** (*int*) – the desired number of columns.
- **rows** (*int*) – the desired number of rows.

Returns a list of *Rect* objects of equal size, whose union equals *rect*. Here is the layout of a 3x4 table created by `cell = fitz.make_table(rect, cols=4, rows=3)`:

cell[0][0]	cell[0][1]	cell[0][2]	cell[0][3]
cell[1][0]	cell[1][1]	cell[1][2]	cell[1][3]
cell[2][0]	cell[2][1]	cell[2][2]	cell[2][3]

planishLine (*p1*, *p2*)*(New in version 1.16.2)*

Return a matrix which maps the line from *p1* to *p2* to the x-axis such that *p1* will become (0,0) and *p2* a point with the same distance to (0,0).

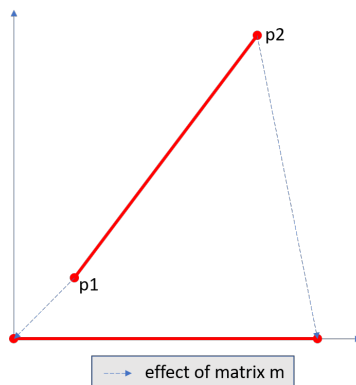
Parameters

- **p1** (*point_like*) – starting point of the line.
- **p2** (*point_like*) – end point of the line.

Return type *Matrix***Returns**

a matrix which combines a rotation and a translation:

```
p1 = fitz.Point(1, 1)
p2 = fitz.Point(4, 5)
abs(p2 - p1) # distance of points
5.0
m = fitz.planishLine(p1, p2)
p1 * m
Point(0.0, 0.0)
p2 * m
Point(5.0, -5.960464477539063e-08)
# distance of the resulting points
abs(p2 * m - p1 * m)
5.0
```

**paperSizes**

A dictionary of pre-defines paper formats. Used as basis for *PaperSize* ().

fitz_fontdescriptors*(New in v1.17.5)*

A dictionary of usable fonts from repository `pymupdf-fonts`. Items are keyed by their reserved fontname and provide information like this:

```
In [2]: fitz.fitz_fontdescriptors.keys()
Out[2]: dict_keys(['figbo', 'figo', 'figbi', 'figit', 'fimbo', 'fimo',
'spacembo', 'spacembi', 'spacemit', 'spacemo', 'math', 'music', 'symbol1
↪',
'symbol2'])
In [3]: fitz.fitz_fontdescriptors["fimo"]
Out[3]:
{'name': 'Fira Mono Regular',
'size': 125712,
'mono': True,
'bold': False,
'italic': False,
'serif': True,
'glyphs': 1485}
```

getPDFnow()

Convenience function to return the current local timestamp in PDF compatible format, e.g. `D:20170501121525-04'00'` for local datetime May 1, 2017, 12:15:25 in a timezone 4 hours westward of the UTC meridian.

Return type str

Returns current local PDF timestamp.

getTextlength (*text*, *fontname*="helv", *fontsize*=11, *encoding*=TEXT_ENCODING_LATIN)*(New in version 1.14.7)*

Calculate the length of text on output with a given **builtin** font, *fontsize* and *encoding*.

Parameters

- **text** (*str*) – the text string.
- **fontname** (*str*) – the fontname. Must be one of either the *PDF Base 14 Fonts* or the CJK fonts, identified by their “reserved” fontnames (see table in `:meth:Page.insertFont`).
- **fontsize** (*float*) – size of the font.
- **encoding** (*int*) – the encoding to use. Besides 0 = Latin, 1 = Greek and 2 = Cyrillic (Russian) are available. Relevant for Base-14 fonts “Helvetica”, “Courier” and “Times” and their variants only. Make sure to use the same value as in the corresponding text insertion.

Return type float

Returns the length in points the string will have (e.g. `when used in Page.insertText()`).

Note: This function will only do the calculation – it won’t insert font or text.

Warning: If you use this function to determine the required rectangle width for the (*Page* or *Shape*) *insertTextbox* methods, be aware that they calculate on a **by-character level**. Because of rounding effects, this will mostly lead to a slightly larger number: $\text{sum}([\text{fitz.getTextlength}(c) \text{ for } c \text{ in } \textit{text}]) > \text{fitz.getTextlength}(\textit{text})$. So either (1) do the same, or (2) use something like $\text{fitz.getTextlength}(\textit{text} + \text{""})$ for your calculation.

getPDFstr (*text*)

Make a PDF-compatible string: if the text contains code points $\text{ord}(c) > 255$, then it will be converted to UTF-16BE with BOM as a hexadecimal character string enclosed in “<>” brackets like `<feff...>`. Otherwise, it will return the string enclosed in (round) brackets, replacing any characters outside the ASCII range with some special code. Also, every “(”, “)” or backslash is escaped with an additional backslash.

Parameters *text* (*str*) – the object to convert

Return type *str*

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

ImageProperties (*stream*)

(New in version 1.14.14)

Return a number of basic properties for an image.

Parameters *stream* (*bytes|bytearray|BytesIO|file*) – an image either in memory or an **opened** file. A memory resident image maybe any of the formats *bytes*, *bytearray* or *io.BytesIO*.

Returns

a dictionary with the following keys (an empty dictionary for any error):

Key	Value
width	(int) width in pixels
height	(int) height in pixels
colorspace	(int) colorspace.n (e.g. 3 = RGB)
bpc	(int) bits per component (usually 8)
format	(int) image format in <i>range(15)</i>
ext	(str) image file extension indicating the format
size	(int) length of the image in bytes

Example:

```
>>> fitz.ImageProperties(open("img-clip.jpg", "rb"))
{'bpc': 8, 'format': 9, 'colorspace': 3, 'height': 325, 'width': 244,
 → 'ext': 'jpeg', 'size': 14161}
>>>
```

ConversionHeader (*text*, *filename="UNKNOWN"*)

Return the header string required to make a valid document out of page text outputs.

Parameters

- **output** (*str*) – type of document. Use the same as the output parameter of *getText()*.
- **filename** (*str*) – optional arbitrary name to use in output types “json” and “xml”.

Return type *str*

ConversionTrailer (*output*)

Return the trailer string required to make a valid document out of page text outputs. See *Page.getText()* for an example.

Parameters **output** (*str*) – type of document. Use the same as the output parameter of *getText()*.

Return type *str*

Document.delete_object (*xref*)

PDF only: Delete an object given by its cross reference number.

Parameters **xref** (*int*) – the cross reference number. Must be within the document’s valid *xref* range.

Warning: Only use with extreme care: this may make the PDF unreadable.

Document.del_xml_metadata ()

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.set_xml_metadata (*xml*)

Store data as the document’s XML Metadata. Correct format is up to the programmer – there is no checking. Any previous such data are overwritten.

Parameters **xml** (*str*) – The data to store

Document.pdf_trailer (*compressed=False*)

(*New in version 1.14.9*)

Return the trailer of the PDF (UTF-8), which is usually located at the PDF file’s end. If not a PDF or the PDF has no trailer (because of irrecoverable errors), *None* is returned.

Parameters **compressed** (*bool*) – (*ew in version 1.14.14*) whether to generate a compressed output or one with nice indentations to ease reading (default).

Returns a string with the PDF trailer information. This is the analogous method to *Document.xref_object()* except that the trailer has no identifying *xref* number. As can be seen here, the trailer object points to other important objects:

```

>>> doc=fitz.open("adobe.pdf")
>>> # compressed output
>>> print(doc.pdf_trailer(True))
<</Size 334093/Prev 25807185/XRefStm 186352/Root 333277 0 R/Info 109959
↳0 R
/ID[ (\227\366/gx\016ds\244\207\326\261\305\376u)
(H\323\177\346\371pkF\243\262\375\346\325\002) ]>>
>>> # non-compressed output:
>>> print(doc.pdf_trailer(False))
<<
  /Size 334093
  /Prev 25807185
  /XRefStm 186352
  /Root 333277 0 R
  /Info 109959 0 R
  /ID [ (\227\366/gx\016ds\244\207\326\261\305\376u)
↳(H\323\177\346\371pkF\243\262\375\346\325\002) ]
>>

```

Note: MuPDF is capable of recovering from a number of damages a PDF may have. This includes re-generating a trailer, where the end of a file has been lost (e.g. because of incomplete downloads). If however *None* is returned for a PDF, then the recovery mechanisms did not work and you should check for any error messages: `print(fitz.TOOLS.mupdf_warnings())`.

`Document._make_page_map()`

Create an internal array of page numbers, which significantly speeds up page lookup (`Document.loadPage()`). If this array exists, finding a page object will be up to two times faster. Functions which change the PDF's page layout (copy, delete, move, select pages) will destroy this array again.

`Document.xml_metadata_xref()`

Return the XML-based metadata *xref* of the PDF if present – also refer to `Document._delXmlMetadata()`. You can use it to retrieve the content via `Document.xrefStream()` and then work with it using some XML software.

Return type `int`

Returns *xref* of PDF file level XML metadata – or 0 if none exists.

`Document._getPageObjNumber(pno)`

or

`Document.page_xref(pno)`

Return the *xref* and generation number for a given page.

Parameters `pno` (*int*) – Page number (zero-based).

Return type `list`

Returns *xref* and generation number of page *pno* as a list [*xref*, *gen*].

Document.**pdf_catalog** ()

Return the *xref* of the PDF catalog.

Return type int

Returns *xref* of the PDF catalog – a central *dictionary* pointing to many other PDF information.

Page.**run** (*dev*, *transform*)

Run a page through a device.

Parameters

- **dev** (*Device*) – Device, obtained from one of the *Device* constructors.
 - **transform** (*Matrix*) – Transformation to apply to the page. Set it to *Identity* if no transformation is desired.
-

Page.**wrap_contents** ()

Put string pair “q” / “Q” before, resp. after a page’s */Contents* object(s) to ensure that any “geometry” changes are **local** only.

Use this method as an alternative, minimalistic version of *Page.clean_contents* (). Its advantage is a small footprint in terms of processing time and impact on the data size of incremental saves.

Page.**is_wrapped**

Indicate whether *Page.wrap_contents* () may be required for object insertions in standard PDF geometry. Please note that this is a quick, basic check only: a value of *False* may still be a false alarm.

Page.**getTextBlocks** (*flags=None*)

Deprecated wrapper for *TextPage.extractBLOCKS* ().

Page.**getTextWords** (*flags=None*)

Deprecated wrapper for *TextPage.extractWORDS* ().

Page.**getDisplayList** ()

Run a page through a list device and return its display list.

Return type *DisplayList*

Returns the display list of the page.

Page.**__getContent**s ()

Return a list of *xref* numbers of *contents* objects belonging to the page.

Return type list

Returns a list of *xref* integers.

Each page may have zero to many associated contents objects (*streams*) which contain some operator syntax describing what appears where and how on the page (like text or images, etc. See the *Adobe PDF References*, chapter “Operator Summary”, page 985). This function only enumerates the number(s) of such objects. To get the actual stream source, use function `Document.xrefStream()` with one of the numbers in this list. Use `Document.updateStream()` to replace the content.

Page.`__setContents(xref)`

PDF only: Set a given object (identified by its *xref*) as the page’s one and only *contents* object. Useful for joining multiple *contents* objects as in the following snippet:

```
>>> c = b""
>>> xreflist = page._getContents()
>>> for xref in xreflist:
>>>     c += doc.xrefStream(xref)
>>> doc.updateStream(xreflist[0], c)
>>> page._setContents(xreflist[0])
>>> # doc.save(..., garbage=1) will remove the unused objects
```

Parameters *xref* (*int*) – the cross reference number of a *contents* object. An exception is raised if outside the valid *xref* range or not a stream object.

Page.`clean_contents(sanitize=True)`

(Changed in v1.17.6)

PDF only: Clean and concatenate all *contents* objects associated with this page. “Cleaning” includes syntactical corrections, standardizations and “pretty printing” of the contents stream. Discrepancies between *contents* and *resources* objects will also be corrected if *sanitize* is true. See `Page.getContents()` for more details.

Changed in version 1.16.0 Annotations are no longer implicitly cleaned by this method. Use `Annot._cleanContents()` separately.

Parameters *sanitize* (*bool*) – (new in v1.17.6) if true, synchronization between resources and their actual use in the contents object is synchronized. For example, if a font is not actually used for any text of the page, then it will be deleted from the `/Resources/Font` object.

Warning: This is a complex function which may generate large amounts of new data and render old data unused. It is **not recommended** using it together with the **incremental save** option. Also note that the resulting singleton new `/Contents` object is **uncompressed**. So you should save to a **new file** using options “`deflate=True, garbage=3`”.

Page.`readContents()`

New in version 1.17.0. Return the concatenation of all *contents* objects associated with the page – without cleaning or otherwise modifying them. Use this method whenever you need to parse this source in its entirety without having to bother how many separate contents objects exist.

Annot.`clean_contents` (*sanitize=True*)

Clean the *contents* streams associated with the annotation. This is the same type of action which `Page.clean_contents()` performs – just restricted to this annotation.

Document.`getCharWidths` (*xref=0, limit=256*)

Return a list of character glyphs and their widths for a font that is present in the document. A font must be specified by its PDF cross reference number *xref*. This function is called automatically from `Page.insertText()` and `Page.insertTextbox()`. So you should rarely need to do this yourself.

Parameters

- **xref** (*int*) – cross reference number of a font embedded in the PDF. To find a font *xref*, use e.g. `doc.getPageFontList(pno)` of page number *pno* and take the first entry of one of the returned list entries.
- **limit** (*int*) – limits the number of returned entries. The default of 256 is enforced for all fonts that only support 1-byte characters, so-called “simple fonts” (checked by this method). All *PDF Base 14 Fonts* are simple fonts.

Return type list

Returns a list of *limit* tuples. Each character *c* has an entry (*g, w*) in this list with an index of `ord(c)`. Entry *g* (integer) of the tuple is the glyph id of the character, and float *w* is its normalized width. The actual width for some fontsize can be calculated as $w * \text{fontsize}$. For simple fonts, the *g* entry can always be safely ignored. In all other cases *g* is the basis for graphically representing *c*.

This function calculates the pixel width of a string called *text*:

```
def pixlen(text, widthlist, fontsize):
    try:
        return sum([widthlist[ord(c)] for c in text]) * fontsize
    except IndexError:
        m = max([ord(c) for c in text])
        raise ValueError("max. code point found: %i, increase limit" % m)
```

Document.`xref_object` (*xref, compressed=False*)

Return the string (“source code”) representing an arbitrary object. For *stream* objects, only the non-stream part is returned. To get the stream data, use `Document.xrefStream()`.

Parameters

- **xref** (*int*) – *xref* number.
- **compressed** (*bool*) – (*new in version 1.14.14*) whether to generate a compressed output or one with nice indentations to ease reading or parsing (default).

Return type string

Returns the string defining the object identified by *xref*. Example:

```
>>> doc = fitz.open("Adobe PDF Reference 1-7.pdf") # the PDF
>>> page = doc[100] # some page in it
>>> print(doc.xref_object(page.xref, compressed=True))
<</CropBox[0 0 531 666]/Annots[4795 0 R 4794 0 R 4793 0 R 4792 0 R 4797
↪0 R 4796 0 R]
```

(continues on next page)

(continued from previous page)

```

/Parent 109820 0 R/StructParents 941/Contents 229 0 R/Rotate 0/
↳MediaBox[0 0 531 666]
/Resources<</Font<</T1_0 3914 0 R/T1_1 3912 0 R/T1_2 3957 0 R/T1_3 3913_
↳0 R/T1_4 4576 0 R
/T1_5 3931 0 R/T1_6 3944 0 R>>/ProcSet[/PDF/Text]/ExtGState<</GS0 333283_
↳0 R>>>>
/Type/Page>>
>>> print (doc.xref_object (page.xref, compressed=False))
<<
/CropBox [ 0 0 531 666 ]
/Annots [ 4795 0 R 4794 0 R 4793 0 R 4792 0 R 4797 0 R 4796 0 R ]
/Parent 109820 0 R
/StructParents 941
/Contents 229 0 R
/Rotate 0
/MediaBox [ 0 0 531 666 ]
/Resources <<
  /Font <<
    /T1_0 3914 0 R
    /T1_1 3912 0 R
    /T1_2 3957 0 R
    /T1_3 3913 0 R
    /T1_4 4576 0 R
    /T1_5 3931 0 R
    /T1_6 3944 0 R
  >>
  /ProcSet [ /PDF /Text ]
  /ExtGState <<
    /GS0 333283 0 R
  >>
>>
/Type /Page
>>

```

Document.**isStream**(*xref*)
(New in version 1.14.14)

PDF only: Check whether the object represented by *xref* is a *stream* type. Return is *False* if not a PDF or if the number is outside the valid *xref* range.

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

Returns *True* if the object definition is followed by data wrapped in keyword pair *stream*, *endstream*.

Document.**get_new_xref**()

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.**xref_length**()

Return length of *xref* table.

Return type int

Returns the number of entries in the *xref* table.

`Document._getOLRootNumber()`

Return *xref* number of the `/Outlines` root object (this is **not** the first outline entry!). If this object does not exist, a new one will be created.

Return type int

Returns *xref* number of the `/Outlines` root object.

`Document.extractImage(xref)`

PDF Only: Extract data and meta information of an image stored in the document. The output can directly be used to be stored as an image file, as input for PIL, *Pixmap* creation, etc. This method avoids using pixmaps wherever possible to present the image in its original format (e.g. as JPEG).

Parameters *xref* (*int*) – *xref* of an image object. If this is not in *range(1, doc.xrefLength())*, or the object is no image or other errors occur, *None* is returned and no exception is raised.

Return type dict

Returns

a dictionary with the following keys

- *ext* (*str*) image type (e.g. `'jpeg'`), usable as image file extension
- *smask* (*int*) *xref* number of a stencil (`/SMask`) image or zero
- *width* (*int*) image width
- *height* (*int*) image height
- *colorspace* (*int*) the image's *colorspace.n* number.
- *cs-name* (*str*) the image's *colorspace.name*.
- *xres* (*int*) resolution in x direction. Please also see *resolution*.
- *yres* (*int*) resolution in y direction. Please also see *resolution*.
- *image* (*bytes*) image data, usable as image file content

```
>>> d = doc.extractImage(1373)
>>> d
{'ext': 'png', 'smask': 2934, 'width': 5, 'height': 629, 'colorspace': 3,
 → 'xres': 96,
  'yres': 96, 'cs-name': 'DeviceRGB',
  'image': b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x00\x05\ ...'}
>>> imgout = open("image." + d["ext"], "wb")
>>> imgout.write(d["image"])
102
>>> imgout.close()
```

Note: There is a functional overlap with `pix = fitz.Pixmap(doc, xref)`, followed by a `pix.getPNGData()`. Main differences are that `extractImage`, (1) does not always deliver PNG image formats, (2) is **very** much faster with non-PNG images, (3) usually results in much less disk storage

for extracted images, (4) returns *None* in error cases (generates no exception). Look at the following example images within the same PDF.

- xref 1268 is a PNG – Comparable execution time and identical output:

```
In [23]: %timeit pix = fitz.Pixmap(doc, 1268);pix.getPNGData()
10.8 ms ± 52.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops_
↳each)
In [24]: len(pix.getPNGData())
Out[24]: 21462

In [25]: %timeit img = doc.extractImage(1268)
10.8 ms ± 86 µs per loop (mean ± std. dev. of 7 runs, 100 loops_
↳each)
In [26]: len(img["image"])
Out[26]: 21462
```

- xref 1186 is a JPEG – `Document.extractImage()` is **many times faster** and produces a **much smaller output** (2.48 MB vs. 0.35 MB):

```
In [27]: %timeit pix = fitz.Pixmap(doc, 1186);pix.getPNGData()
341 ms ± 2.86 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [28]: len(pix.getPNGData())
Out[28]: 2599433

In [29]: %timeit img = doc.extractImage(1186)
15.7 µs ± 116 ns per loop (mean ± std. dev. of 7 runs, 100000 loops_
↳each)
In [30]: len(img["image"])
Out[30]: 371177
```

`Document.extractFont(xref, info_only=False)`

PDF Only: Return an embedded font file’s data and appropriate file extension. This can be used to store the font as an external file. The method does not throw exceptions (other than via checking for PDF and valid *xref*).

Parameters

- **xref** (*int*) – PDF object number of the font to extract.
- **info_only** (*bool*) – only return font information, not the buffer. To be used for information-only purposes, avoids allocation of large buffer areas.

Return type tuple

Returns

a tuple (*basename*, *ext*, *subtype*, *buffer*), where *ext* is a 3-byte suggested file extension (*str*), *basename* is the font’s name (*str*), *subtype* is the font’s type (e.g. “Type1”) and *buffer* is a bytes object containing the font file’s content (or *b”*”). For possible extension values and their meaning see *Font File Extensions*. Return details on error:

- (“”, “”, “”, *b”*”) – invalid xref or xref is not a (valid) font object.
- (*basename*, “n/a”, “Type1”, *b”*”) – *basename* is one of the *PDF Base 14 Fonts*, which cannot be extracted.

Example:

```
>>> # store font as an external file
>>> name, ext, buffer = doc.extractFont(4711)
>>> # assuming buffer is not None:
>>> ofile = open(name + "." + ext, "wb")
>>> ofile.write(buffer)
>>> ofile.close()
```

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

Document.**FontInfos**

Contains following information for any font inserted via `Page.insertFont()` in **this** session of PyMuPDF:

- `xref (int)` – XREF number of the `/Type/Font` object.
- `info (dict)` – detail font information with the following keys:
 - `name (str)` – name of the basefont
 - `idx (int)` – index number for multi-font files
 - `type (str)` – font type (like “TrueType”, “Type0”, etc.)
 - `ext (str)` – extension to be used, when font is extracted to a file (see [Font File Extensions](#)).
 - `glyphs (list)` – list of glyph numbers and widths (filled by textinsertion methods).

Return type list

8.2 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 | ...`.

8.3 Working together: DisplayList and TextPage

Here are some instructions on how to use these classes together.

In some situations, performance improvements may be achievable, when you fall back to the detail level explained here.

8.3.1 Create a DisplayList

A *DisplayList* represents an interpreted document page. Methods for pixmap creation, text extraction and text search are – behind the curtain – all using the page’s display list to perform their tasks. If a page must be rendered several times (e.g. because of changed zoom levels), or if text search and text extraction should both be performed, overhead can be saved, if the display list is created only once and then used for all other tasks.

```
>>> dl = page.getDisplayList()           # create the display list
```

You can also create display lists for many pages “on stack” (in a list), may be during document open, during idling times, or you store it when a page is visited for the first time (e.g. in GUI scripts).

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

8.3.2 Generate Pixmap

The following creates a Pixmap from a *DisplayList*. Parameters are the same as for *Page.getPixmap()*.

```
>>> pix = dl.getPixmap()                 # create the page's pixmap
```

The execution time of this statement may be up to 50% shorter than that of *Page.getPixmap()*.

8.3.3 Perform Text Search

With the display list from above, we can also search for text.

For this we need to create a *TextPage*.

```
>>> tp = dl.getTextPage()                 # display list from above
>>> rlist = tp.search("needle")           # look up "needle" locations
>>> for r in rlist:                        # work with the found locations, e.g.
    pix.invertIRect(r.irect)              # invert colors in the rectangles
```

8.3.4 Extract Text

With the same *TextPage* object from above, we can now immediately use any or all of the 5 text extraction methods.

Note: Above, we have created our text page without argument. This leads to a default argument of 3 (ligatures and white-space are preserved), 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
```

8.3.5 Further Performance improvements

8.3.5.1 Pixmap

As explained in the *Page* chapter:

If you do not need transparency set *alpha = 0* when creating pixmaps. This will save 25% memory (if RGB, the most common case) and possibly 5% execution time (depending on the GUI software).

8.3.5.2 TextPage

If you do not need images extracted alongside the text of a page, you can set the following option:

```
>>> flags = fitz.TEXT_PRESERVE_LIGATURES | fitz.TEXT_PRESERVE_WHITESPACE
>>> tp = dl.getTextPage(flags)
```

This will save ca. 25% overall execution time for the HTML, XHTML and JSON text extractions and **hugely** reduce the amount of storage (both, memory and disk space) if the document is graphics oriented.

If you however do need images, use a value of 7 for flags:

```
>>> flags = fitz.TEXT_PRESERVE_LIGATURES | fitz.TEXT_PRESERVE_WHITESPACE | fitz.TEXT_
↳PRESERVE_IMAGES
```

matrix_like

A Python sequence of 6 numbers.

rect_like

A Python sequence of 4 numbers.

irect_like

A Python sequence of 4 integers.

point_like

A Python sequence of 2 numbers.

quad_like

A Python sequence of 4 *point_like* items.

inheritable

A number of values in a PDF can be specified once and then be inherited by objects further down in a parent-child relationship. The mediabox (physical size) of pages can for example be specified in some node(s) of the *pagetree* and will then be taken as value for all *kids*, which do not specify their own value.

MediaBox

A PDF array of 4 floats specifying a physical page size – (*inheritable*).

CropBox

A PDF array of 4 floats specifying a page's visible area – (*inheritable*). This value is **not affected** if the page is rotated. In contrast to the page rectangle, *Page.rect*, the top-left corner of the cropbox may or may not be $(0, 0)$.

catalog

A central PDF *dictionary* – also called the “root” – containing document-wide parameters and pointers to many other information.

contents

“A **content stream** is a PDF *object* with an attached *stream*, whose data consists of a sequence of instructions describing the graphical elements to be painted on a page.” (*Adobe PDF References* p. 151). For an overview of the mini-language used in these streams see chapter “Operator Summary” on page 985 of the *Adobe PDF References*. A PDF *page* can have none to many contents objects. If it has none, the page is empty

(but still may show annotations). If it has several, they will be interpreted in sequence as if their instructions had been present in one such object (i.e. like in a concatenated string). It should be noted that there are more stream object types which use the same syntax: e.g. appearance dictionaries associated with annotations and Form XObjects.

resources

A *dictionary* containing references to any resources (like images or fonts) required by a PDF *page* (required, inheritable, *Adobe PDF References* p. 145) and certain other objects (Form XObjects). This dictionary appears as a sub-dictionary in the object definition under the key */Resources*. Being an inheritable object type, there may exist “parent” resources for all pages or certain subsets of pages.

dictionary

A PDF *object* type, which is somewhat comparable to the same-named Python notion: “A dictionary object is an associative table containing pairs of objects, known as the dictionary’s entries. The first element of each entry is the key and the second element is the value. The key must be a name (...). The value can be any kind of object, including another dictionary. A dictionary entry whose value is null (...) is equivalent to an absent entry.” (*Adobe PDF References* p. 59).

Dictionaries are the most important *object* type in PDF. Here is an example (describing a *page*):

```
<<
/Contents 40 0 R           % value: an indirect object
/Type/Page                % value: a name object
/MediaBox[0 0 595.32 841.92] % value: an array object
/Rotate 0                 % value: a number object
/Parent 12 0 R           % value: an indirect object
/Resources<<              % value: a dictionary object
  /ExtGState<</R7 26 0 R>>
  /Font<<
    /R8 27 0 R/R10 21 0 R/R12 24 0 R/R14 15 0 R
    /R17 4 0 R/R20 30 0 R/R23 7 0 R /R27 20 0 R
  >>
  /ProcSet[/PDF/Text]     % value: array of two name objects
>>
/Annots[55 0 R]          % value: array, one entry (indirect object)
>>
```

Contents, *Type*, *MediaBox*, etc. are **keys**, *40 0 R*, *Page*, *[0 0 595.32 841.92]*, etc. are the respective **values**. The strings “<<” and “>>” are used to enclose object definitions.

This example also shows the syntax of **nested** dictionary values: *Resources* has an object as its value, which in turn is a dictionary with keys like *ExtGState* (with the value <</R7 26 0 R>>, which is another dictionary), etc.

page

A PDF page is a *dictionary* object which defines one page in a PDF, see *Adobe PDF References* p. 145.

pagetree

“The pages of a document are accessed through a structure known as the page tree, which defines the ordering of pages in the document. The tree structure allows PDF consumer applications, using only limited memory, to quickly open a document containing thousands of pages. The tree contains nodes of two types: intermediate nodes, called page tree nodes, and leaf nodes, called page objects.” (*Adobe PDF References* p. 143).

While it is possible to list all page references in just one array, PDFs with many pages are often created using *balanced tree* structures (“page trees”) for faster access to any single page. In relation to the total number of pages, this can reduce the average page access time by page number from a linear to some logarithmic order of magnitude.

For fast page access, MuPDF can use its own array in memory – independently from what may or may not be present in the document file. This array is indexed by page number and therefore much faster than even the

access via a perfectly balanced page tree.

object

Similar to Python, PDF supports the notion *object*, which can come in eight basic types: boolean values, integer and real numbers, strings, names, arrays, dictionaries, streams, and the null object (*Adobe PDF References* p. 51). Objects can be made identifiable by assigning a label. This label is then called *indirect* object. PyMuPDF supports retrieving definitions of indirect objects via their cross reference number via `Document.xrefObject()`.

stream

A PDF *object* type which is followed by a sequence of bytes, similar to a Python *string* or rather *bytes*. “However, a PDF application can read a stream incrementally, while a string must be read in its entirety. Furthermore, a stream can be of unlimited length, whereas a string is subject to an implementation limit. For this reason, objects with potentially large amounts of data, such as images and page descriptions, are represented as streams.” “A stream consists of a *dictionary* followed by zero or more bytes bracketed between the keywords *stream* and *endstream*”:

```

nnn 0 obj
<<
  dictionary definition
>>
stream
(zero or more bytes)
endstream
endobj

```

See *Adobe PDF References* p. 60. PyMuPDF supports retrieving stream content via `Document.xrefStream()`. Use `Document.isStream()` to determine whether an object is of stream type.

unitvector

A mathematical notion meaning a vector of norm (“length”) 1 – usually the Euclidean norm is implied. In PyMuPDF, this term is restricted to *Point* objects, see `Point.unit`.

xref

Abbreviation for cross-reference number: this is an integer unique identification for objects in a PDF. There exists a cross-reference table (which may physically consist of several separate segments) in each PDF, which stores the relative position of each object for quick lookup. The cross-reference table is one entry longer than the number of existing object: item zero is reserved and must not be used in any way. Many PyMuPDF classes have an *xref* attribute (which is zero for non-PDFs), and one can find out the total number of objects in a PDF via `Document.xrefLength() - 1`.

resolution

Images and *Pixmap* objects may contain resolution information provided as “dots per inch”, dpi, in each direction (horizontal and vertical). When MuPDF reads an image from a file or from a PDF object, it will parse this information and put it in `Pixmap.xres`, `Pixmap.yres`, respectively. When it finds not meaningful information in the input (like non-positive values or values exceeding 4800), it will use “sane” defaults instead. The usual default value is 96, but it may also be 72 in some cases (e.g. for JPX images).

OCPD

Optional content properties dictionary - a sub *dictionary* of the PDF *catalog*. The central place to store optional content information, which is identified by the key `/OCProperties`. This dictionary has two required and one optional entry: (1) `/OCGs`, required, an array listing all optional content groups, (2) `/D`, required, the default optional content configuration dictionary (OCCD), (3) `/Configs`, optional, an array of alternative OCCDs.

OCCD

Optional content configuration dictionary - a PDF *dictionary* inside the PDF *OCPD*. It stores a setting of ON / OFF states of OCGs and how they are presented to a PDF viewer program. Selecting a configuration is quick way to achieve temporary mass visibility state changes. After opening a PDF, the `/D` configuration of the *OCPD*

is always activated. Viewer should offer a way to switch between the */D*, or one of the optional configurations contained in array */Configs*.

OCG

Optional content group – a *dictionary* object used to control the visibility of other PDF objects like images or annotations. Independently on which page they are defined, objects with the same OCG can simultaneously be shown or hidden by setting their OCG to ON or OFF. This can be achieved via the user interface provided by many PDF viewers (Adobe Acrobat), or programmatically.

OCMD

Optional content membership dictionary – a *dictionary* object which can be used like an *OCG*: it has a visibility state. The visibility of an OCMD is **computed**: it is a logical expression, which uses the state of one or more OCGs to produce a boolean value. The expression's result is interpreted as ON (true) or OFF (false).

Constants and Enumerations

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

10.1 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

10.2 Document Permissions

Code	Permitted Action
PDF_PERM_PRINT	Print the document
PDF_PERM_MODIFY	Modify the document’s contents
PDF_PERM_COPY	Copy or otherwise extract text and graphics
PDF_PERM_ANNOTATE	Add or modify text annotations and interactive form fields
PDF_PERM_FORM	Fill in forms and sign the document
PDF_PERM_ACCESSIBILITY	Obsolete, always permitted
PDF_PERM_ASSEMBLE	Insert, rotate, or delete pages, bookmarks, thumbnail images
PDF_PERM_PRINT_HQ	High quality printing

10.3 PDF encryption method codes

Code	Meaning
PDF_ENCRYPT_KEEP	do not change
PDF_ENCRYPT_NONE	remove any encryption
PDF_ENCRYPT_RC4_40	RC4 40 bit
PDF_ENCRYPT_RC4_128	RC4 128 bit
PDF_ENCRYPT_AES_128	<i>Advanced Encryption Standard 128 bit</i>
PDF_ENCRYPT_AES_256	<i>Advanced Encryption Standard 256 bit</i>
PDF_ENCRYPT_UNKNOWN	unknown

10.4 Font File Extensions

The table show file extensions you should use when extracting fonts from a PDF file.

Ext	Description
ttf	TrueType font
pfa	Postscript for ASCII font (various subtypes)
cff	Type1C font (compressed font equivalent to Type1)
cid	character identifier font (postscript format)
otf	OpenType font
n/a	built-in font (<i>PDF Base 14 Fonts</i> or CJK: cannot be extracted)

10.5 Text Alignment

TEXT_ALIGN_LEFT

0 – align left.

TEXT_ALIGN_CENTER

1 – align center.

TEXT_ALIGN_RIGHT

2 – align right.

TEXT_ALIGN_JUSTIFY

3 – align justify.

10.6 Preserve Text Flags

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

TEXT_PRESERVE_LIGATURES

1 – If set, ligatures are passed through to the application in their original form. Otherwise ligatures are expanded into their constituent parts, e.g. the ligature ffi is expanded into three eparate characters f, f and i.

TEXT_PRESERVE_WHITESPACE

2 – If set, whitespace is passed through to the application in its original form. Otherwise any type of horizontal whitespace (including horizontal tabs) will be replaced with space characters of variable width.

TEXT_PRESERVE_IMAGES

4 – If set, then images will be stored in the structured text structure.

TEXT_INHIBIT_SPACES

8 – If set, we will not try to add missing space characters where there are large gaps between characters.

TEXT_DEHYPHENATE

16 – Ignore hyphens at line ends and join with next line. Used mainly with search function

TEXT_PRESERVE_SPANS

32 – Generate a new line for every span. Not used in PyMuPDF.

10.7 Link Destination Kinds

Possible values of `linkDest.kind` (link destination kind). For details consult *Adobe PDF References*, 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_NAMED

4 – points to a named location.

Return type int

LINK_GOTOR

5 – Points to a place in another PDF document.

Return type int

10.8 Link Destination Flags

Note: The rightmost byte of this integer is a bit field, so test the truth of these bits with the `&` operator.

LINK_FLAG_L_VALID

1 (bit 0) Top left x value is valid

Return type bool

LINK_FLAG_T_VALID

2 (bit 1) Top left y value is valid

Return type bool

LINK_FLAG_R_VALID

4 (bit 2) Bottom right x value is valid

Return type bool

LINK_FLAG_B_VALID

8 (bit 3) Bottom right y value is valid

Return type bool

LINK_FLAG_FIT_H

16 (bit 4) Horizontal fit

Return type bool**LINK_FLAG_FIT_V**

32 (bit 5) Vertical fit

Return type bool**LINK_FLAG_R_IS_ZOOM**

64 (bit 6) Bottom right x is a zoom figure

Return type bool

10.9 Annotation Related Constants

See chapter 8.4.5, pp. 615 of the *Adobe PDF References* for details.

10.9.1 Annotation Types

These identifiers also cover **links** and **widgets**: the PDF specification technically handles them all in the same way, whereas **MuPDF** (and **PyMuPDF**) treats them as three basically different types of objects.

```

PDF_ANNOT_TEXT 0
PDF_ANNOT_LINK 1 # <=== Link object in PyMuPDF
PDF_ANNOT_FREE_TEXT 2
PDF_ANNOT_LINE 3
PDF_ANNOT_SQUARE 4
PDF_ANNOT_CIRCLE 5
PDF_ANNOT_POLYGON 6
PDF_ANNOT_POLY_LINE 7
PDF_ANNOT_HIGHLIGHT 8
PDF_ANNOT_UNDERLINE 9
PDF_ANNOT_SQUIGGLY 10
PDF_ANNOT_STRIKE_OUT 11
PDF_ANNOT_REDACT 12
PDF_ANNOT_STAMP 13
PDF_ANNOT_CARET 14
PDF_ANNOT_INK 15
PDF_ANNOT_POPUP 16
PDF_ANNOT_FILE_ATTACHMENT 17
PDF_ANNOT_SOUND 18
PDF_ANNOT_MOVIE 19
PDF_ANNOT_RICH_MEDIA 20
PDF_ANNOT_WIDGET 21 # <=== Widget object in PyMuPDF
PDF_ANNOT_SCREEN 22
PDF_ANNOT_PRINTER_MARK 23
PDF_ANNOT_TRAP_NET 24
PDF_ANNOT_WATERMARK 25
PDF_ANNOT_3D 26
PDF_ANNOT_PROJECTION 27
PDF_ANNOT_UNKNOWN -1

```

10.9.2 Annotation Flag Bits

```
PDF_ANNOT_IS_INVISIBLE 1 << (1-1)
PDF_ANNOT_IS_HIDDEN 1 << (2-1)
PDF_ANNOT_IS_PRINT 1 << (3-1)
PDF_ANNOT_IS_NO_ZOOM 1 << (4-1)
PDF_ANNOT_IS_NO_ROTATE 1 << (5-1)
PDF_ANNOT_IS_NO_VIEW 1 << (6-1)
PDF_ANNOT_IS_READ_ONLY 1 << (7-1)
PDF_ANNOT_IS_LOCKED 1 << (8-1)
PDF_ANNOT_IS_TOGGLE_NO_VIEW 1 << (9-1)
PDF_ANNOT_IS_LOCKED_CONTENTS 1 << (10-1)
```

10.9.3 Annotation Line Ending Styles

```
PDF_ANNOT_LE_NONE 0
PDF_ANNOT_LE_SQUARE 1
PDF_ANNOT_LE_CIRCLE 2
PDF_ANNOT_LE_DIAMOND 3
PDF_ANNOT_LE_OPEN_ARROW 4
PDF_ANNOT_LE_CLOSED_ARROW 5
PDF_ANNOT_LE_BUTT 6
PDF_ANNOT_LE_R_OPEN_ARROW 7
PDF_ANNOT_LE_R_CLOSED_ARROW 8
PDF_ANNOT_LE_SLASH 9
```

10.10 Widget Constants

10.10.1 Widget Types (*field_type*)

```
PDF_WIDGET_TYPE_UNKNOWN 0
PDF_WIDGET_TYPE_BUTTON 1
PDF_WIDGET_TYPE_CHECKBOX 2
PDF_WIDGET_TYPE_COMBOBOX 3
PDF_WIDGET_TYPE_LISTBOX 4
PDF_WIDGET_TYPE_RADIOBUTTON 5
PDF_WIDGET_TYPE_SIGNATURE 6
PDF_WIDGET_TYPE_TEXT 7
```

10.10.2 Text Widget Subtypes (*text_format*)

```
PDF_WIDGET_TX_FORMAT_NONE 0
PDF_WIDGET_TX_FORMAT_NUMBER 1
PDF_WIDGET_TX_FORMAT_SPECIAL 2
PDF_WIDGET_TX_FORMAT_DATE 3
PDF_WIDGET_TX_FORMAT_TIME 4
```

10.10.3 Widget flags (*field_flags*)

Common to all field types:

```
PDF_FIELD_IS_READ_ONLY 1
PDF_FIELD_IS_REQUIRED 1 << 1
PDF_FIELD_IS_NO_EXPORT 1 << 2
```

Text widgets:

```
PDF_TX_FIELD_IS_MULTILINE 1 << 12
PDF_TX_FIELD_IS_PASSWORD 1 << 13
PDF_TX_FIELD_IS_FILE_SELECT 1 << 20
PDF_TX_FIELD_IS_DO_NOT_SPELL_CHECK 1 << 22
PDF_TX_FIELD_IS_DO_NOT_SCROLL 1 << 23
PDF_TX_FIELD_IS_COMB 1 << 24
PDF_TX_FIELD_IS_RICH_TEXT 1 << 25
```

Button widgets:

```
PDF_BTN_FIELD_IS_NO_TOGGLE_TO_OFF 1 << 14
PDF_BTN_FIELD_IS_RADIO 1 << 15
PDF_BTN_FIELD_IS_PUSHBUTTON 1 << 16
PDF_BTN_FIELD_IS_RADIOS_IN_UNISON 1 << 25
```

Choice widgets:

```
PDF_CH_FIELD_IS_COMBO 1 << 17
PDF_CH_FIELD_IS_EDIT 1 << 18
PDF_CH_FIELD_IS_SORT 1 << 19
PDF_CH_FIELD_IS_MULTI_SELECT 1 << 21
PDF_CH_FIELD_IS_DO_NOT_SPELL_CHECK 1 << 22
PDF_CH_FIELD_IS_COMMIT_ON_SEL_CHANGE 1 << 26
```

10.11 PDF Standard Blend Modes

For an explanation see *Adobe PDF References*, page 520:

```
PDF_BM_Color "Color"
PDF_BM_ColorBurn "ColorBurn"
PDF_BM_ColorDodge "ColorDodge"
PDF_BM_Darken "Darken"
PDF_BM_Difference "Difference"
PDF_BM_Exclusion "Exclusion"
PDF_BM_HardLight "HardLight"
PDF_BM_Hue "Hue"
PDF_BM_Lighten "Lighten"
PDF_BM_Luminosity "Luminosity"
PDF_BM_Multiply "Multiply"
PDF_BM_Normal "Normal"
PDF_BM_Overlay "Overlay"
PDF_BM_Saturation "Saturation"
PDF_BM_Screen "Screen"
PDF_BM_SoftLight "Softlight"
```

10.12 Stamp Annotation Icons

MuPDF has defined the following icons for **rubber stamp** annotations:

```
STAMP_Approved 0
STAMP_AsIs 1
STAMP_Confidential 2
STAMP_Departmental 3
STAMP_Experimental 4
STAMP_Expired 5
STAMP_Final 6
STAMP_ForComment 7
STAMP_ForPublicRelease 8
STAMP_NotApproved 9
STAMP_NotForPublicRelease 10
STAMP_Sold 11
STAMP_TopSecret 12
STAMP_Draft 13
```

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 memorable 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.

11.1 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")
(0.94111764705882353, 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
```

(continues on next page)

(continued from previous page)

```
[('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)] ...
```

11.2 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.



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.

12.1 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. *pdfwr* 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 *pdfiopng* of Xpdf – see part 3 of this chapter.

Tool	Description
PyMuPDF	tool of this manual, appearing as “fitz” in reports
pdfwr	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")
```

pdfwr:

```
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):

Runtimes	Tool			
	fitz	pdfrw	pdftk	PyPDF2
Adobe.pdf	4,96	20,72	136,34	683,27
Evolution.pdf	0,40	0,41	1,22	0,94
PyMuPDF.pdf	0,04	0,19	1,03	0,97
sdw_2015_01.pdf	0,19	1,19	6,13	6,49
sdw_2015_02.pdf	0,23	1,52	7,74	7,02
sdw_2015_03.pdf	0,39	2,76	13,39	12,67
sdw_2015_04.pdf	0,25	2,14	8,55	7,50
sdw_2015_05.pdf	0,29	1,71	8,92	7,99
sdw_2015_06.pdf	0,53	3,30	16,05	15,56
sdw_2015_07.pdf	0,33	2,17	10,65	10,81
sdw_2015_08.pdf	0,29	2,01	9,65	9,39
sdw_2015_09.pdf	0,36	2,49	11,48	10,97
sdw_2015_10.pdf	0,27	1,87	3,31	6,74
sdw_2015_11.pdf	1,47	12,79	40,18	62,44
sdw_2015_12.pdf	0,39	2,21	10,40	10,19
Total Times	10,40	57,46	285,04	852,96

Time Ratios			
1,00	5,52	27,40	81,98
	1,00	4,96	14,84
		1,00	2,99
			1,00

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

Runtimes	Tool			
	fitz	pdfwr	pdftk	PyPDF2
Evolution.pdf	0,40	0,41	1,22	0,94
PyMuPDF.pdf	0,04	0,19	1,03	0,97
sdw_2015_01.pdf	0,19	1,19	6,13	6,49
sdw_2015_02.pdf	0,23	1,52	7,74	7,02
sdw_2015_03.pdf	0,39	2,76	13,39	12,67
sdw_2015_04.pdf	0,25	2,14	8,55	7,50
sdw_2015_05.pdf	0,29	1,71	8,92	7,99
sdw_2015_06.pdf	0,53	3,30	16,05	15,56
sdw_2015_07.pdf	0,33	2,17	10,65	10,81
sdw_2015_08.pdf	0,29	2,01	9,65	9,39
sdw_2015_09.pdf	0,36	2,49	11,48	10,97
sdw_2015_10.pdf	0,27	1,87	3,31	6,74
sdw_2015_11.pdf	1,47	12,79	40,18	62,44
sdw_2015_12.pdf	0,39	2,21	10,40	10,19
Gesamtergebnis	5,44	36,75	148,70	169,69

Time Ratios			
1,00	6,75	27,32	31,18
	1,00	4,05	4,62
		1,00	1,14
			1,00

PyMuPDF is by far the fastest: on average 4.5 times faster than the second best (the pure Python tool pdfwr, **chapeau pdfwr!**), 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 pdfwr! 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 pdfwr 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 pdfwr 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.

12.2 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](#).

12.3 Part 3: Image Rendering

We have tested rendering speed of MuPDF against the *pdftopng.exe*, a command line 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.

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)?

13.1 General structure of a *TextPage*

TextPage is one of PyMuPDF's classes. It is normally created behind the curtain, when *Page* text extraction methods are used, but it is also available directly. In any case, an intermediate class, *DisplayList* must be created first (display lists contain interpreted pages, they also provide the input for *Pixmap* creation). Information contained in a *TextPage* has the following hierarchy. Other than its name suggests, images may optionally also be part of a text page:

```
<page>
  <text block>
    <line>
      <span>
        <char>
  <image block>
    <img>
```

A **text page** consists of blocks (= roughly paragraphs).

A **block** consists of either lines and their characters, or an image.

A **line** consists of spans.

A **span** consists of adjacent characters with identical font properties: name, size, flags and color.

13.2 Plain Text

Function `TextPage.extractText()` (or `Page.getText("text")`) extracts a page's plain **text in original order** as specified by the creator of the document (which may not equal a natural reading order).

An example output:

```
>>> print(page.getText("text"))
Some text on first page.
```

13.3 BLOCKS

Function `TextPage.extractBLOCKS()` (or `Page.getText("blocks")`) extracts a page's text blocks as a list of items like:

```
(x0, y0, x1, y1, "lines in block", block_type, block_no)
```

Where the first 4 items are the float coordinates of the block's bbox. The lines within each block are concatenated by a new-line character.

This is a high-speed method with enough information to re-arrange the page's text in natural reading order where required.

Example output:

```
>>> print(page.getText("blocks"))
[(50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375,
'Some text on first page.', 0, 0)]
```

13.4 WORDS

Function `TextPage.extractWORDS()` (or `Page.getText("words")`) extracts a page's text **words** as a list of items like:

```
(x0, y0, x1, y1, "word", block_no, line_no, word_no)
```

Where the first 4 items are the float coordinates of the words's bbox. The last three integers provide some more information on the word's whereabouts.

This is a high-speed method with enough information to extract text contained in a given rectangle.

Example output:

```
>>> for word in page.getText("words"):
    print(word)
(50.0, 88.17500305175781, 78.73200225830078, 103.28900146484375,
'Some', 0, 0, 0)
(81.79000091552734, 88.17500305175781, 99.5219955444336, 103.28900146484375,
'text', 0, 0, 1)
(102.57999420166016, 88.17500305175781, 114.8119888305664, 103.28900146484375,
'on', 0, 0, 2)
(117.86998748779297, 88.17500305175781, 135.5909881591797, 103.28900146484375,
'first', 0, 0, 3)
```

(continues on next page)

(continued from previous page)

```
(138.64898681640625, 88.17500305175781, 166.1709747314453, 103.28900146484375,
'page.', 0, 0, 4)
```

13.5 HTML

`TextPage.extractHTML()` (or `Page.getText("html")`) output fully reflects the structure of the page's `TextPage` – much like DICT / JSON below. This includes images, font information and text positions. If wrapped in HTML header and trailer code, it can readily be displayed by an internet browser. Our above example:

```
>>> for line in page.getText("html").splitlines():
      print(line)

<div id="page0" style="position:relative;width:300pt;height:350pt;
background-color:white">
<p style="position:absolute;white-space:pre;margin:0;padding:0;top:88pt;
left:50pt"><span style="font-family:Helvetica,sans-serif;
font-size:11pt">Some text on first page.</span></p>
</div>
```

13.6 Controlling Quality of HTML Output

While HTML output has improved a lot in MuPDF v1.12.0, it is not yet bug-free: we have found problems in the areas **font support** and **image positioning**.

- HTML text contains references to the fonts used of the original document. If these are not known to the browser (a fat chance!), it will replace them with others; the results will probably 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()
pos1 = 0
font_serif = "font-family:Times"
font_sans = "font-family:Helvetica"
font_mono = "font-family:Courier"
found_one = False

while True:
    pos0 = otext.find("font-family:", pos1)
    if pos0 < 0:
        break
    pos1 = otext.find(";", pos0)
    test = otext[pos0 : pos1]
```

(continues on next page)

(continued from previous page)

```

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
    posl = 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!")

```

13.7 DICT (or JSON)

`TextPage.extractDICT()` (or `Page.getText("dict")`) output fully reflects the structure of a `TextPage` and provides image content and position details (*bbox* – boundary boxes in pixel units) for every block and line. This information can be used to present text in another reading order if required (e.g. from top-left to bottom-right). Images are stored as *bytes* (*bytearray* in Python 2) for DICT output and base64 encoded strings for JSON output.

For a visuallization of the dictionary structure have a look at *Dictionary Structure of extractDICT() and extractRAW-DICT()*.

Here is how this looks like:

```

{
  "width": 300.0,
  "height": 350.0,
  "blocks": [{
    "type": 0,
    "bbox": [50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375],
    "lines": [{
      "wmode": 0,
      "dir": [1.0, 0.0],
      "bbox": [50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375],
      "spans": [{
        "size": 11.0,
        "flags": 0,
        "font": "Helvetica",
        "color": 0,
        "text": "Some text on first page.",
        "bbox": [50.0, 88.17500305175781, 166.1709747314453, 103.
↪28900146484375]
      }]
    }]
  }]
}

```

13.8 RAWDICT

`TextPage.extractRAWDICT()` (or `Page.getText("rawdict")`) is an **information superset of DICT** and takes the detail level one step deeper. It looks exactly like the above, except that the “*text*” items (*string*) are replaced by “*chars*” items (*list*). Each “*chars*” entry is a character *dict*. For example, here is what you would see in place of item “*text*”: “*Text in black color.*” above:

```
"chars": [{
  "origin": [50.0, 100.0],
  "bbox": [50.0, 88.17500305175781, 57.336997985839844, 103.28900146484375],
  "c": "S"
}, {
  "origin": [57.33700180053711, 100.0],
  "bbox": [57.33700180053711, 88.17500305175781, 63.4530029296875, 103.
↪28900146484375],
  "c": "o"
}, {
  "origin": [63.4530029296875, 100.0],
  "bbox": [63.4530029296875, 88.17500305175781, 72.61600494384766, 103.
↪28900146484375],
  "c": "m"
}, {
  "origin": [72.61600494384766, 100.0],
  "bbox": [72.61600494384766, 88.17500305175781, 78.73200225830078, 103.
↪28900146484375],
  "c": "e"
}, {
  "origin": [78.73200225830078, 100.0],
  "bbox": [78.73200225830078, 88.17500305175781, 81.79000091552734, 103.
↪28900146484375],
  "c": " "
}
< ... deleted ... >
}, {
  "origin": [163.11297607421875, 100.0],
  "bbox": [163.11297607421875, 88.17500305175781, 166.1709747314453, 103.
↪28900146484375],
  "c": "."
}],
```

13.9 XML

The `TextPage.extractXML()` (or `Page.getText("xml")`) version extracts text (no images) with the detail level of RAWDICT:

```
>>> for line in page.getText("xml").splitlines():
      print(line)

<page id="page0" width="300" height="350">
<block bbox="50 88.175 166.17098 103.289">
<line bbox="50 88.175 166.17098 103.289" wmode="0" dir="1 0">
<font name="Helvetica" size="11">
<char quad="50 88.175 57.336999 88.175 50 103.289 57.336999 103.289" x="50"
y="100" color="#000000" c="S"/>
<char quad="57.337 88.175 63.453004 88.175 57.337 103.289 63.453004 103.289" x="57.337
↪"
```

(continues on next page)

(continued from previous page)

```

y="100" color="#000000" c="o"/>
<char quad="63.453004 88.175 72.616008 88.175 63.453004 103.289 72.616008 103.289" x=
↳ "63.453004"
y="100" color="#000000" c="m"/>
<char quad="72.616008 88.175 78.732 88.175 72.616008 103.289 78.732 103.289" x="72.
↳ 616008"
y="100" color="#000000" c="e"/>
<char quad="78.732 88.175 81.79 88.175 78.732 103.289 81.79 103.289" x="78.732"
y="100" color="#000000" c=" " />

... deleted ...

<char quad="163.11298 88.175 166.17098 88.175 163.11298 103.289 166.17098 103.289" x=
↳ "163.11298"
y="100" color="#000000" c="." />
</font>
</line>
</block>
</page>

```

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

13.10 XHTML

`TextPage.extractXHTML()` (or `Page.getText("xhtml")`) is a variation of TEXT but in HTML format, containing the bare text and images (“semantic” output):

```

<div id="page0">
<p>Some text on first page.</p>
</div>

```

13.11 Text Extraction Flags Defaults

(New in version 1.16.2) Method `Page.getText()` supports a keyword parameter `flags` (*int*) to control the amount and the quality of extracted data. The following table shows the defaults settings (flags parameter omitted or `None`) for each extraction variant. If you specify flags with a value other than `None`, be aware that you must set **all desired** options. A description of the respective bit settings can be found in [Preserve Text Flags](#).

Indicator	text	html	xhtml	xml	dict	rawdict	words	blocks
preserve ligatures	1	1	1	1	1	1	1	1
preserve whitespace	1	1	1	1	1	1	1	1
preserve images	n/a	1	1	n/a	1	1	n/a	0
inhibit spaces	0	0	0	0	0	0	0	0
dehyphenate	0	0	0	0	0	0	0	0

- “**json**” is handled exactly like “**dict**” and is hence left out.
- An “n/a” specification means a value of 0 and setting this bit never has any effect on the output (but an adverse effect on performance).

- If you are not interested in images when using an output variant which includes them by default, then by all means set the respective bit off: You will experience a better performance and much lower space requirements.

To show the effect of `TEXT_INHIBIT_SPACES` have a look at this example:

```
>>> print(page.getText("text"))
H a l l o !
M o r e t e x t
i s f o l l o w i n g
i n E n g l i s h
. . . l e t ' s s e e
w h a t h a p p e n s .
>>> print(page.getText("text", flags=fitz.TEXT_INHIBIT_SPACES))
Hallo!
More text
is following
in English
... let's see
what happens.
>>>
```

13.12 Performance

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

Note: Especially images have a **very significant** impact. Make sure to exclude them (via the `flags` parameter) whenever you do not need them. To process the below mentioned 2'700 total pages with default flags settings required 160 seconds across all extraction methods. When all images were excluded, less than 50% of that time (77 seconds) were needed.

To begin with, all methods are **very fast** in relation to other products out there in the market. In terms of processing speed, we are not aware of a faster (free) tool. Even the most detailed method, RAWDICT, processes all 1'310 pages of the *Adobe PDF References* in less than 5 seconds (simple text needs less than 2 seconds here).

The following table shows average relative speeds (“RSpeed”, baseline 1.00 is TEXT), taken across ca. 1400 text-heavy and 1300 image-heavy pages.

Method	RSpeed	Comments	no images
TEXT	1.00	no images, plain text, line breaks	1.00
BLOCKS	1.00	image bboxes (only), block level text with bboxes, line breaks	1.00
WORDS	1.02	no images, word level text with bboxes	1.02
XML	2.72	no images, char level text, layout and font details	2.72
XHTML	3.32	base64 images, span level text, no layout info	1.00
HTML	3.54	base64 images, span level text, layout and font details	1.01
DICT	3.93	binary images, span level text, layout and font details	1.04
RAWDICT	4.50	binary images, char level text, layout and font details	1.68

As mentioned: when excluding all images (last column), the relative speeds are changing drastically: except RAW-DICT and XML, the other methods are almost equally fast, and RAWDICT requires 40% less execution time than the **now slowest XML**.

Look at chapter **Appendix 1** for more performance information.

Appendix 3: Considerations on Embedded Files

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

14.1 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 References*).

In many aspects, this is comparable to concepts also found in ZIP files or the OLE technique in MS Windows. PDF embedded files do, however, *not* support directory structures as does the ZIP format. An embedded file can in turn contain embedded files itself.

Advantages of this concept are that embedded files are under the PDF umbrella, benefitting from its permissions / password protection and integrity aspects: all data, which a PDF may reference or even may be dependent on, can be bundled into it and so form a single, consistent unit of information.

In addition to embedded files, PDF 1.7 adds *collections* to its support range. This is an advanced way of storing and presenting meta information (i.e. arbitrary and extensible properties) of embedded files.

14.2 MuPDF Support

After adding initial support for collections (portfolios) and */EmbeddedFiles* in MuPDF version 1.11, this support was dropped again in version 1.15.

As a consequence, the cli utility *mutool* no longer offers access to embedded files.

PyMuPDF – having implemented an */EmbeddedFiles* API in response in its version 1.11.0 – was therefore forced to change gears starting with its version 1.16.0 (we never published a MuPDF v1.15.x compatible PyMuPDF).

We are now maintaining our own code basis supporting embedded files. This code makes use of basic MuPDF dictionary and array functions only.

14.3 PyMuPDF Support

We continue to support the full old API with respect to embedded files – with only minor, cosmetic changes.

There even also is a new function, which delivers a list of all names under which embedded data are resgistered in a PDF, `Document.embeddedFileNames()`.

Appendix 4: Assorted Technical Information

15.1 PDF Base 14 Fonts

The following 14 builtin font names **must be supported by every PDF viewer** application. They are available as a dictionary, which maps their full names and their abbreviations in lower case to the full font basename. Wherever a **fontname** must be provided in PyMuPDF, any **key or value** from the dictionary may be used:

```
In [2]: fitz.Base14_fontdict
Out[2]:
{'courier': 'Courier',
 'courier-oblique': 'Courier-Oblique',
 'courier-bold': 'Courier-Bold',
 'courier-boldoblique': 'Courier-BoldOblique',
 'helvetica': 'Helvetica',
 'helvetica-oblique': 'Helvetica-Oblique',
 'helvetica-bold': 'Helvetica-Bold',
 'helvetica-boldoblique': 'Helvetica-BoldOblique',
 'times-roman': 'Times-Roman',
 'times-italic': 'Times-Italic',
 'times-bold': 'Times-Bold',
 'times-bolditalic': 'Times-BoldItalic',
 'symbol': 'Symbol',
 'zapfdingbats': 'ZapfDingbats',
 'helv': 'Helvetica',
 'heit': 'Helvetica-Oblique',
 'hebo': 'Helvetica-Bold',
 'hebi': 'Helvetica-BoldOblique',
 'cour': 'Courier',
 'coit': 'Courier-Oblique',
 'cobo': 'Courier-Bold',
 'cobi': 'Courier-BoldOblique',
 'tiro': 'Times-Roman',
 'tibo': 'Times-Bold',
 'tiit': 'Times-Italic',
```

(continues on next page)

(continued from previous page)

```
'tibi': 'Times-BoldItalic',  
'symb': 'Symbol',  
'zadb': 'ZapfDingbats'}
```

In contrast to their obligation, not all PDF viewers support these fonts correctly and completely – this is especially true for Symbol and ZapfDingbats. Also, the glyph (visual) images will be specific to every reader.

To see how these fonts can be used – including the **CJK built-in** fonts – look at the table in `Page.insertFont()`.

15.2 Adobe PDF References

This PDF Reference manual published by Adobe is frequently quoted throughout this documentation. It can be viewed and downloaded from [here](#).

There is a newer version of this, which can be found [here](#). Redaction annotations are an example contained in this one, but not in the earlier version.

15.3 Using Python Sequences as Arguments in PyMuPDF

When PyMuPDF objects and methods require a Python **list** of numerical values, other Python **sequence types** are also allowed. Python classes are said to implement the **sequence protocol**, if they have a `__getitem__()` method.

This basically means, you can interchangeably use Python *list* or *tuple* or even *array.array*, *numpy.array* and *bytearray* types in these cases.

For example, specifying a sequence “s” in any of the following ways

- `s = [1, 2]`
- `s = (1, 2)`
- `s = array.array("i", (1, 2))`
- `s = numpy.array((1, 2))`
- `s = bytearray((1, 2))`

will make it usable in the following example expressions:

- `fitz.Point(s)`
- `fitz.Point(x, y) + s`
- `doc.select(s)`

Similarly with all geometry objects *Rect*, *IRect*, *Matrix* and *Point*.

Because all PyMuPDF geometry classes themselves are special cases of sequences, they (with the exception of *Quad* – see below) can be freely used where numerical sequences can be used, e.g. as arguments for functions like `list()`, `tuple()`, `array.array()` or `numpy.array()`. Look at the following snippet to see this work.

```

>>> import fitz, array, numpy as np
>>> m = fitz.Matrix(1, 2, 3, 4, 5, 6)
>>>
>>> list(m)
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
>>>
>>> tuple(m)
(1.0, 2.0, 3.0, 4.0, 5.0, 6.0)
>>>
>>> array.array("f", m)
array('f', [1.0, 2.0, 3.0, 4.0, 5.0, 6.0])
>>>
>>> np.array(m)
array([1., 2., 3., 4., 5., 6.])

```

Note: *Quad* is a Python sequence object as well and has a length of 4. Its items however are *point_like* – not numbers. Therefore, the above remarks do not apply.

15.4 Ensuring Consistency of Important Objects in PyMuPDF

PyMuPDF is a Python binding for the C library MuPDF. While a lot of effort has been invested by MuPDF’s creators to approximate some sort of an object-oriented behavior, they certainly could not overcome basic shortcomings of the C language in that respect.

Python on the other hand implements the OO-model in a very clean way. The interface code between PyMuPDF and MuPDF consists of two basic files: *fitz.py* and *fitz_wrap.c*. They are created by the excellent SWIG tool for each new version.

When you use one of PyMuPDF’s objects or methods, this will result in execution of some code in *fitz.py*, which in turn will call some C code compiled with *fitz_wrap.c*.

Because SWIG goes a long way to keep the Python and the C level in sync, everything works fine, if a certain set of rules is being strictly followed. For example: **never access a *Page* object**, after you have closed (or deleted or set to *None*) the owning *Document*. Or, less obvious: **never access a page or any of its children (links or annotations) after you have executed one of the document methods *select()*, *deletePage()*, *insertPage()* ... and more.**

But just no longer accessing invalidated objects is actually not enough: They should rather be actively deleted entirely, to also free C-level resources (meaning allocated memory).

The reason for these rules lies in the fact that there is a hierarchical 2-level one-to-many relationship between a document and its pages and also between a page and its links / annotations. To maintain a consistent situation, any of the above actions must lead to a complete reset – in **Python and, synchronously, in C**.

SWIG cannot know about this and consequently does not do it.

The required logic has therefore been built into PyMuPDF itself in the following way.

1. If a page “loses” its owning document or is being deleted itself, all of its currently existing annotations and links will be made unusable in Python, and their C-level counterparts will be deleted and deallocated.
2. If a document is closed (or deleted or set to *None*) or if its structure has changed, then similarly all currently existing pages and their children will be made unusable, and corresponding C-level deletions will take place. “Structure changes” include methods like *select()*, *deletePage()*, *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.

15.5 Design of Method `Page.showPDFpage()`

15.5.1 Purpose and Capabilities

The method displays an image of a (“source”) page of another PDF document within a specified rectangle of the current (“containing”, “target”) page.

- **In contrast** to `Page.insertImage()`, this display is vector-based and hence remains accurate across zooming levels.
- **Just like** `Page.insertImage()`, the size of the display is adjusted to the given rectangle.

The following variations of the display are currently supported:

- Bool parameter `keep_proportion` controls whether to maintain the aspect ratio (default) or not.
- Rectangle parameter `clip` restricts the visible part of the source page rectangle. Default is the full page.
- float `rotation` rotates the display by an arbitrary angle (degrees). If the angle is not an integer multiple of 90, only 2 of the 4 corners may be positioned on the target border if also `keep_proportion` is true.
- Bool parameter `overlay` controls whether to put the image on top (foreground, default) of current page content or not (background).

Use cases include (but are not limited to) the following:

1. “Stamp” a series of pages of the current document with the same image, like a company logo or a watermark.
2. Combine arbitrary input pages into one output page to support “booklet” or double-sided printing (known as “4-up”, “n-up”).
3. Split up (large) input pages into several arbitrary pieces. This is also called “posterization”, because you e.g. can split an A4 page horizontally and vertically, print the 4 pieces enlarged to separate A4 pages, and end up with an A2 version of your original page.

15.5.2 Technical Implementation

This is done using PDF “**Form XObjects**”, see section 4.9 on page 355 of *Adobe PDF References*. On execution of a `Page.showPDFpage(rect, src, pno, ...)`, the following things happen:

1. The `resources` and `contents` objects of page `pno` in document `src` are copied over to the current document, jointly creating a new **Form XObject** with the following properties. The PDF `xref` number of this object is returned by the method.
 - a. `/BBox` equals `/Mediabox` of the source page
 - b. `/Matrix` equals the identity matrix `[1 0 0 1 0 0]`
 - c. `/Resources` equals that of the source page. This involves a “deep-copy” of hierarchically nested other objects (including fonts, images, etc.). The complexity involved here is covered by MuPDF’s grafting¹ technique functions.

¹ MuPDF supports “deep-copying” objects between PDF documents. To avoid duplicate data in the target, it uses so-called “graftmaps”, like a form of scratchpad: for each object to be copied, its `xref` number is looked up in the graftmap. If found, copying is skipped. Otherwise, the new `xref` is recorded and the copy takes place. PyMuPDF makes use of this technique in two places so far: `Document.insertPDF()` and `Page.showPDFpage()`. This process is fast and very efficient, because it prevents multiple copies of typically large and frequently referenced data, like images and fonts. However, you may still want to consider using garbage collection (option 4) in any of the following cases:

1. The target PDF is not new / empty: grafting does not check for resource types that already existed (e.g. images, fonts) in the target document
2. Using `Page.showPDFpage()` for more than one source document: each grafting occurs **within one source** PDF only, not across multiple.

- d. This is a stream object type, and its stream is an exact copy of the combined data of the source page's */Contents* objects.

This step is only executed once per shown source page. Subsequent displays of the same page only create pointers (done in next step) to this object.

2. A second **Form XObject** is then created which the target page uses to invoke the display. This object has the following properties:
 - a. */BBox* equals the */CropBox* of the source page (or *clip*).
 - b. */Matrix* represents the mapping of */BBox* to the target rectangle.
 - c. */XObject* references the previous XObject via the fixed name *fullpage*.
 - d. The stream of this object contains exactly one fixed statement: */fullpage Do*.
3. The *resources* and *contents* objects of the target page are now modified as follows.
 - a. Add an entry to the */XObject* dictionary of */Resources* with the name *fzFrm<n>* (with *n* chosen such that this entry is unique on the page).
 - b. Depending on *overlay*, prepend or append a new object to the page's */Contents* array, containing the statement *q /fzFrm<n> Do Q*.

15.6 Redirecting Error and Warning Messages

Since MuPDF version 1.16 error and warning messages can be redirected via an official plugin.

PyMuPDF will put error messages to *sys.stderr* prefixed with the string "mupdf:". Warnings are internally stored and can be accessed via *fitz.TOOLS.mupdf_warnings()*. There also is a function to empty this store.

16.1 Changes in Version 1.18.5

Apart from several fixes, this version also focusses on several minor, but important feature improvements. Among the latter is a more precise computation of proper line heights and insertion points for writing / inserting text. As opposed to using font-agnostic constants, these values are now taken from the font's properties.

Also note that this is the first version which does no longer provide pregenerated wheels for Python versions older than 3.6. PIP also discontinues support for these by end of this year 2020.

- **Fixed** issue #771. By using “small glyph heights” option, the full page text can be extracted.
- **Fixed** issue #768.
- **Fixed** issue #750.
- **Fixed** issue #739. The “dict”, “rawdct” and corresponding JSON output variants now have two new *span* keys: “ascender” and “descender”. These floats represent special font properties which can be used to compute bboxes of spans or characters of **exactly fontsize height** (as opposed to the default line height). An example algorithm is shown in section “Span Dictionary” [here](#). Also improved the detection and correction of ill-specified ascender / descender values encountered in some fonts.
- **Added** a new, experimental `Tools.set_small_glyph_heights()` – also in response to issue #739. This method sets or unsets a global parameter to **always compute bboxes with fontsize height**. If “on”, text searching and all text extractions will returned rectangles, bboxes and quads with a smaller height.
- **Fixed** issue #728.
- **Changed** fill color logic of ‘Polyline’ annotations: this parameter now only pertains to line end symbols – the annotation itself can no longer have a fill color. Also addresses issue #727.
- **Changed** `Page.getImageBbox()` to also compute the bbox if the image is contained in an XObject.
- **Changed** `Shape.insertTextbox()`, resp. `Page.insertTextbox()`, resp. `TextWriter.fillTextbox()` to respect font's properties “ascender” / “descender” when computing line height and insertion point. This should no longer lead to line overlaps for multi-line output. These methods used to ignore font specifics and used constant values instead.

16.2 Changes in Version 1.18.4

This version adds several features to support PDF Optional Content. Among other things, this includes OCMDs (Optional Content Membership Dictionaries) with the full scope of “*visibility expressions*” (PDF key /VE), text insertions (including the *TextWriter* class) and drawings.

- **Fixed** issue #727. Freetext annotations now support an uncolored rectangle when `fill_color=None`.
- **Fixed** issue #726. UTF-8 encoding errors are now handled for HTML / XML `Page.getText()` output.
- **Fixed** issue #724. Empty values are no longer stored in the PDF /Info metadata dictionary.
- **Added** new methods `Document.set_oc()` and `Document.get_oc()` to set or get optional content references for **existing** image and form XObjects. These methods are similar to the same-named methods of *Annot*.
- **Added** `Document.set_ocmd()`, `Document.get_ocmd()` for handling OCMDs.
- **Added Optional Content** support for text insertion and drawing.
- **Added** new method `Page.deleteWidget()`, which deletes a form field from a page. This is analogous to deleting annotations.
- **Added** support for Popup annotations. This includes defining the Popup rectangle and setting the Popup to open or closed. Methods / attributes `Annot.set_popup()`, `Annot.set_open()`, `Annot.has_popup`, `Annot.is_open`, `Annot.popup_rect`, `Annot.popup_xref`.

Other changes:

- The **naming of methods and attributes** in PyMuPDF is far from being satisfactory: we have *CamelCases*, *mixedCases* and *lower_case_with_underscores* all over the place. With the *Annot* as the first candidate, we have started an activity to clean this up step by step, converting to lower case with underscores for methods and attributes while keeping UPPERCASE for the constants.
 - Old names will remain available to prevent code breaks, but they will no longer be mentioned in the documentation.
 - New methods and attributes of all classes will be named according to the new standard.

16.3 Changes in Version 1.18.3

As a major new feature, this version introduces support for PDF’s **Optional Content** concept.

- **Fixed** issue #714.
- **Fixed** issue #711.
- **Fixed** issue #707: if a PDF user password, but no owner password is supplied nor present, then the user password is also used as the owner password.
- **Fixed** expand and deflate parameters of methods `Document.save()` and `Document.write()`. Individual image and font compression should now finally work. Addresses issue #713.
- **Added** a support of PDF optional content. This includes several new *Document* methods for inquiring and setting optional content status and adding optional content configurations and groups. In addition, images, form XObjects and annotations now can be bound to optional content specifications. **Resolved** issue #709.

16.4 Changes in Version 1.18.2

This version contains some interesting improvements for text searching: any number of search hits is now returned and the `hit_max` parameter was removed. The new `clip` parameter in addition allows to restrict the search area. Searching now detects hyphenations at line breaks and accordingly finds hyphenated words.

- **Fixed** issue #575: if using `quads=False` in text searching, then overlapping rectangles on the same line are joined. Previously, parts of the search string, which belonged to different “marked content” items, each generated their own rectangle – just as if occurring on separate lines.
- **Added** `Document.isRepaired`, which is true if the PDF was repaired on open.
- **Added** `Document.setXmlMetadata()` which either updates or creates PDF XML metadata. Implements issue #691.
- **Added** `Document.getXmlMetadata()` returns PDF XML metadata.
- **Changed** creation of PDF documents: they will now always carry a PDF identification (/ID field) in the document trailer. Implements issue #691.
- **Changed** `Page.searchFor()`: a new parameter `clip` is accepted to restrict the search to this rectangle. Correspondingly, the attribute `TextPage.rect` is now respected by `TextPage.search()`.
- **Changed** parameter `hit_max` in `Page.searchFor()` and `TextPage.search()` is now obsolete: methods will return all hits.
- **Changed** character **selection criteria** in `Page.getText()`: a character is now considered to be part of a `clip` if its `bbox` is fully contained. Before this, a non-empty intersection was sufficient.
- **Changed** `Document.scrub()` to support a new option `redact_images`. This addresses issue #697.

16.5 Changes in Version 1.18.1

- **Fixed** issue #692. PyMuPDF now detects and recovers from more cyclic resource dependencies in PDF pages and for the first time reports them in the MuPDF warnings store.
- **Fixed** issue #686.
- **Added** opacity options for the `Shape` class: Stroke and fill colors can now be set to some transparency value. This means that all `Page` draw methods, methods `Page.insertText()`, `Page.insertTextbox()`, `Shape.finish()`, `Shape.insertText()`, and `Shape.insertTextbox()` support two new parameters: `stroke_opacity` and `fill_opacity`.
- **Added** new parameter `mask` to `Page.insertImage()` for optionally providing an external image mask. Resolves issue #685.
- **Added** `Annot.soundGet()` for extracting the sound of an audio annotation.

16.6 Changes in Version 1.18.0

This is the first PyMuPDF version supporting MuPDF v1.18. The focus here is on extending PyMuPDF’s own functionality – apart from bug fixing. Subsequent PyMuPDF patches may address features new in MuPDF.

- **Fixed** issue #519. This upstream bug occurred occasionally for some pages only and seems to be fixed now: page layout should no longer be ruined in these cases.
- **Fixed** issue #675.

- Unsuccessful storage allocations should now always lead to exceptions (circumvention of an upstream bug intermittently crashing the interpreter).
- *Pixmap* size is now based on `size_t` instead of `int` in C and should be correct even for extremely large pixmaps.
- **Fixed** issue #668. Specification of dashes for PDF drawing insertion should now correctly reflect the PDF spec.
- **Fixed** issue #669. A major source of memory leakage in `Page.insertPDF()` has been removed.
- **Added** keyword “*images*” to `Page.apply_redactions()` for fine-controlling the handling of images.
- **Added** `Annot.getText()` and `Annot.getTextbox()`, which offer the same functionality as the *Page* versions.
- **Added** key “*number*” to the block dictionaries of `Page.getText()` / `Annot.getText()` for options “*dict*” and “*rawdict*”.
- **Added** `glyph_name_to_unicode()` and `unicode_to_glyph_name()`. Both functions do not really connect to a specific font and are now independently available, too. The data are now based on the [Adobe Glyph List](#).
- **Added** convenience functions `adobe_glyph_names()` and `adobe_glyph_unicodes()` which return the respective available data.
- **Added** `Page.getDrawings()` which returns details of drawing operations on a document page. Works for all document types.
- Improved performance of `Document.insertPDF()`. Multiple object copies are now also suppressed across multiple separate insertions from the same source. This saves time, memory and target file size. Previously this mechanism was only active within each single method execution. The feature can also be suppressed with the new method bool parameter `final=1`, which is the default.
- For PNG images created from pixmaps, the resolution (dpi) is now automatically set from the respective `Pixmap.xres` and `Pixmap.yres` values.

16.7 Changes in Version 1.17.7

- **Fixed** issue #651. An upstream bug causing interpreter crashes in corner case redaction processings was fixed by backporting MuPDF changes from their development repo.
- **Fixed** issue #645. *Pixmap* top-left coordinates can be set (again) by their own method, `Pixmap.setOrigin()`.
- **Fixed** issue #622. `Page.insertImage()` again accepts a `rect_like` parameter.
- **Added** several new methods to improve and speed-up table of contents (TOC) handling. Among other things, TOC items can now be changed or deleted individually – without always replacing the complete TOC. Furthermore, access to some PDF page attributes is now possible without first **loading** the page. This has a very significant impact on the performance of TOC manipulation.
- **Added** an option to `Document.insertPDF()` which allows displaying progress messages. Addresses #640.
- **Added** `Page.getTextbox()` which extracts text contained in a rectangle. In many cases, this should obsolete writing your own script for this type of thing.
- **Added** new `clip` parameter to `Page.getText()` to simplify and speed up text extraction of page sub areas.
- **Added** `TextWriter.appendv()` to add text in **vertical write mode**. Addresses issue #653

16.8 Changes in Version 1.17.6

- **Fixed** issue #605
- **Fixed** issue #600 – text should now be correctly positioned also for pages with a CropBox smaller than Media-Box.
- **Added** text span dictionary key `origin` which contains the lower left coordinate of the first character in that span.
- **Added** attribute `Font.buffer`, a *bytes* copy of the font file.
- **Added** parameter `sanitize` to `Page.cleanContents()`. Allows switching of sanitization, so only syntax cleaning will be done.

16.9 Changes in Version 1.17.5

- **Fixed** issue #561 – second go: certain *TextWriter* usages with many alternating fonts did not work correctly.
- **Fixed** issue #566.
- **Fixed** issue #568.
- **Fixed** – opacity is now correctly taken from the *TextWriter* object, if not given in *TextWriter.writeText()*.
- **Added** a new global attribute `fitz_fontdescriptors`. Contains information about usable fonts from repository `pymupdf-fonts`.
- **Added** `Font.valid_codepoints()` which returns an array of unicode codepoints for which the font has a glyph.
- **Added** option `text_as_path` to `Page.getSVGImage()`. this implements #580. Generates much smaller SVG files with parseable text if set to *False*.

16.10 Changes in Version 1.17.4

- **Fixed** issue #561. Handling of more than 10 *Font* objects on one page should now work correctly.
- **Fixed** issue #562. Annotation pixmaps are no longer derived from the page pixmap, thus avoiding unintended inclusion of page content.
- **Fixed** issue #559. This **MuPDF** bug is being temporarily fixed with a pre-version of MuPDF's next release.
- **Added** utility function `repair_mono_font()` for correcting displayed character spacing for some mono-spaced fonts.
- **Added** utility method `Document.need_appearances()` for fine-controlling Form PDF behavior. Addresses issue #563.
- **Added** utility function `sRGB_to_pdf()` to recover the PDF color triple for a given color integer in sRGB format.
- **Added** utility function `sRGB_to_rgb()` to recover the (R, G, B) color triple for a given color integer in sRGB format.
- **Added** utility function `make_table()` which delivers table cells for a given rectangle and desired numbers of columns and rows.

- **Added** support for optional fonts in repository `pymupdf-fonts`.

16.11 Changes in Version 1.17.3

- **Fixed** an undocumented issue, which prevented fully cleaning a PDF page when using `Page.cleanContents()`.
- **Fixed** issue #540. Text extraction for EPUB should again work correctly.
- **Fixed** issue #548. Documentation now includes `LINK_NAMED`.
- **Added** new parameter to control start of text in `TextWriter.fillTextbox()`. Implements #549.
- **Changed** documentation of `Page.addRedactAnnot()` to explain the usage of non-builtin fonts.

16.12 Changes in Version 1.17.2

- **Fixed** issue #533.
- **Added** options to modify ‘Redact’ annotation appearance. Implements #535.

16.13 Changes in Version 1.17.1

- **Fixed** issue #520.
- **Fixed** issue #525. Vertices for ‘Ink’ annots should now be correct.
- **Fixed** issue #524. It is now possible to query and set rotation for applicable annotation types.

Also significantly improved inline documentation for better support of interactive help.

16.14 Changes in Version 1.17.0

This version is based on MuPDF v1.17. Following are highlights of new and changed features:

- **Added** extended language support for annotations and widgets: a mixture of Latin, Greece, Russian, Chinese, Japanese and Korean characters can now be used in ‘FreeText’ annotations and text widgets. No special arrangement is required to use it.
- Faster page access is implemented for documents supporting a “chapter” structure. This applies to EPUB documents currently. This comes with several new `Document` methods and changes for `Document.loadPage()` and the “indexed” page access `doc[n]`: In addition to specifying a page number as before, a tuple `(chapter, pno)` can be specified to identify the desired page.
- **Changed:** Improved support of redaction annotations: images overlapped by redactions are **permanently modified** by erasing the overlap areas. Also links are removed if overlapped by redactions. This is now fully in sync with PDF specifications.

Other changes:

- **Changed** `TextWriter.writeText()` to support the “*morph*” parameter.
- **Added** methods `Rect.morph()`, `IRect.morph()`, and `Quad.morph()`, which return a new `Quad`.
- **Changed** `Page.addFreetextAnnot()` to support text alignment via a new “*align*” parameter.

- **Fixed** issue #508. Improved image rectangle calculation to hopefully deliver correct values in most if not all cases.
- **Fixed** issue #502.
- **Fixed** issue #500. `Document.convertToPDF()` should no longer cause memory leaks.
- **Fixed** issue #496. Annotations and widgets / fields are now added or modified using the coordinates of the **unrotated page**. This behavior is now in sync with other methods modifying PDF pages.
- **Added** `Page.rotationMatrix` and `Page.derotationMatrix` to support coordinate transformations between the rotated and the original versions of a PDF page.

Potential code breaking changes:

- The private method `Page._getTransformation()` has been removed. Use the public `Page.transformationMatrix` instead.

16.15 Changes in Version 1.16.18

This version introduces several new features around PDF text output. The motivation is to simplify this task, while at the same time offering extending features.

One major achievement is using MuPDF's capabilities to dynamically choosing fallback fonts whenever a character cannot be found in the current one. This seamlessly works for Base-14 fonts in combination with CJK fonts (China, Japan, Korea). So a text may contain **any combination of characters** from the Latin, Greek, Russian, Chinese, Japanese and Korean languages.

- **Fixed** issue #493. `Pixmap(doc, xref)` should now again correctly resemble the loaded image object.
- **Fixed** issue #488. Widget names are now modifyable.
- **Added** new class `Font` which represents a font.
- **Added** new class `TextWriter` which serves as a container for text to be written on a page.
- **Added** `Page.writeText()` to write one or more `TextWriter` objects to the page.

16.16 Changes in Version 1.16.17

- **Fixed** issue #479. PyMuPDF should now more correctly report image resolutions. This applies to both, images (either from images files or extracted from PDF documents) and pixmaps created from images.
- **Added** `Pixmap.setResolution()` which sets the image resolution in x and y directions.

16.17 Changes in Version 1.16.16

- **Fixed** issue #477.
- **Fixed** issue #476.
- **Changed** annotation line end symbol coloring and fixed an error coloring the interior of 'Polyline' / 'Polygon' annotations.

16.18 Changes in Version 1.16.14

- **Changed** text marker annotations to accept parameters beyond just quadrilaterals such that now **text lines between two given points can be marked**.
- **Added** `Document.scrub()` which **removes potentially sensitive data** from a PDF. Implements #453.
- **Added** `Annot.blendMode()` which returns the **blend mode** of annotations.
- **Added** `Annot.setBlendMode()` to set the annotation's blend mode. This resolves issue #416.
- **Changed** `Annot.update()` to accept additional parameters for setting blend mode and opacity.
- **Added** advanced graphics features to **control the anti-aliasing values**, `Tools.set_aa_level()`. Resolves #467
- **Fixed** issue #474.
- **Fixed** issue #466.

16.19 Changes in Version 1.16.13

- **Added** `Document.getPageXObjectList()` which returns a list of **Form XObjects** of the page.
- **Added** `Page.setMediaBox()` for changing the physical PDF page size.
- **Added** `Page` methods which have been internal before: `Page.cleanContents()` (= `Page._cleanContents()`), `Page.getContents()` (= `Page._getContents()`), `Page.getTransformation()` (= `Page._getTransformation()`).

16.20 Changes in Version 1.16.12

- **Fixed** issue #447
- **Fixed** issue #461.
- **Fixed** issue #397.
- **Fixed** issue #463.
- **Added** JavaScript support to PDF form fields, thereby fixing #454.
- **Added** a new annotation method `Annot.delete_responses()`, which removes 'Popup' and response annotations referring to the current one. Mainly serves data protection purposes.
- **Added** a new form field method `Widget.reset()`, which resets the field value to its default.
- **Changed** and extended handling of redactions: images and XObjects are removed if *contained* in a redaction rectangle. Any partial only overlaps will just be covered by the redaction background color. Now an *overlay* text can be specified to be inserted in the rectangle area to **take the place the deleted original** text. This resolves #434.

16.21 Changes in Version 1.16.11

- **Added** Support for redaction annotations via method `Page.addRedactAnnot()` and `Page.apply_redactions()`.

- **Fixed** issue #426 (“PolygonAnnotation in 1.16.10 version”).
- **Fixed** documentation only issues #443 and #444.

16.22 Changes in Version 1.16.10

- **Fixed** issue #421 (“annot.setRect(rect) has no effect on text Annotation”)
- **Fixed** issue #417 (“Strange behavior for page.deleteAnnot on 1.16.9 compare to 1.13.20”)
- **Fixed** issue #415 (“Annot.setOpacity throws mupdf warnings”)
- **Changed** all “add annotation / widget” methods to store a unique name in the */NM* PDF key.
- **Changed** `Annot.setInfo()` to also accept direct parameters in addition to a dictionary.
- **Changed** `Annot.info` to now also show the annotation’s unique id (*/NM* PDF key) if present.
- **Added** `Page.annot_names()` which returns a list of all annotation names (*/NM* keys).
- **Added** `Page.load_annot()` which loads an annotation given its unique id (*/NM* key).
- **Added** `Document.reload_page()` which provides a new copy of a page after finishing any pending updates to it.

16.23 Changes in Version 1.16.9

- **Fixed** #412 (“Feature Request: Allow controlling whether TOC entries should be collapsed”)
- **Fixed** #411 (“Seg Fault with page.firstWidget”)
- **Fixed** #407 (“Annot.setOpacity trouble”)
- **Changed** methods `Annot.setBorder()`, `Annot.setColors()`, `Link.setBorder()`, and `Link.setColors()` to also accept direct parameters, and not just cumbersome dictionaries.

16.24 Changes in Version 1.16.8

- **Added** several new methods to the `Document` class, which make dealing with PDF low-level structures easier. I also decided to provide them as “normal” methods (as opposed to private ones starting with an underscore “_”). These are `Document.xrefObject()`, `Document.xrefStream()`, `Document.xrefStreamRaw()`, `Document.PDFTrailer()`, `Document.PDFCatalog()`, `Document.metadataXML()`, `Document.updateObject()`, `Document.updateStream()`.
- **Added** `Tools.mupdf_disply_errors()` which sets the display of mupdf errors on `sys.stderr`.
- **Added** a commandline facility. This a major new feature: you can now invoke several utility functions via “`python -m fitz ...`”. It should obsolete the need for many of the most trivial scripts. Please refer to *Using fitz as a Module*.

16.25 Changes in Version 1.16.7

Minor changes to better synchronize the binary image streams of `TextPage` image blocks and `Document.extractImage()` images.

- **Fixed** issue #394 (“PyMuPDF Segfaults when using `TOOLS.mupdf_warnings()`”).
- **Changed** redirection of MuPDF error messages: apart from writing them to Python `sys.stderr`, they are now also stored with the MuPDF warnings.
- **Changed** `Tools.mupdf_warnings()` to automatically empty the store (if not deactivated via a parameter).
- **Changed** `Page.getImageBbox()` to return an **infinite rectangle** if the image could not be located on the page – instead of raising an exception.

16.26 Changes in Version 1.16.6

- **Fixed** issue #390 (“Incomplete deletion of annotations”).
- **Changed** `Page.searchFor()` / `Document.searchPageFor()` to also support the `flags` parameter, which controls the data included in a `TextPage`.
- **Changed** `Document.getPageImageList()`, `Document.getPageFontList()` and their `Page` counterparts to support a new parameter `full`. If true, the returned items will contain the `xref` of the `FormXObject` where the font or image is referenced.

16.27 Changes in Version 1.16.5

More performance improvements for text extraction.

- **Fixed** second part of issue #381 (see item in v1.16.4).
- **Added** `Page.getTextPage()`, so it is no longer required to create an intermediate display list for text extractions. Page level wrappers for text extraction and text searching are now based on this, which should improve performance by ca. 5%.

16.28 Changes in Version 1.16.4

- **Fixed** issue #381 (“TextPage.extractDICT ... failed ... after upgrading ... to 1.16.3”)
- **Added** method `Document.pages()` which delivers a generator iterator over a page range.
- **Added** method `Page.links()` which delivers a generator iterator over the links of a page.
- **Added** method `Page.annots()` which delivers a generator iterator over the annotations of a page.
- **Added** method `Page.widgets()` which delivers a generator iterator over the form fields of a page.
- **Changed** `Document.isFormPDF` to now contain the number of widgets, and `False` if not a PDF or this number is zero.

16.29 Changes in Version 1.16.3

Minor changes compared to version 1.16.2. The code of the “dict” and “rawdict” variants of `Page.getText()` has been ported to C which has greatly improved their performance. This improvement is mostly noticeable with text-oriented documents, where they now should execute almost two times faster.

- **Fixed** issue #369 (“mupdf: cmsCreateTransform failed”) by removing ICC colorspace support.

- **Changed** `Page.getText()` to accept additional keywords “blocks” and “words”. These will deliver the results of `Page.getTextBlocks()` and `Page.getTextWords()`, respectively. So all text extraction methods are now available via a uniform API. Correspondingly, there are now new methods `TextPage.extractBLOCKS()` and `TextPage.extractWords()`.
- **Changed** `Page.getText()` to default bit indicator `TEXT_INHIBIT_SPACES` to **off**. Insertion of additional spaces is **not suppressed** by default.

16.30 Changes in Version 1.16.2

- **Changed** text extraction methods of `Page` to allow detail control of the amount of extracted data.
- **Added** `planishLine()` which maps a given line (defined as a pair of points) to the x-axis.
- **Fixed** an issue (w/o Github number) which brought down the interpreter when encountering certain non-UTF-8 encodable characters while using `Page.getText()` with the “dict” option.
- **Fixed** issue #362 (“Memory Leak with `getText(‘rawDICT’)`”).

16.31 Changes in Version 1.16.1

- **Added** property `Quad.isConvex` which checks whether a line is contained in the quad if it connects two points of it.
- **Changed** `Document.insertPDF()` to now allow dropping or including links and annotations independently during the copy. Fixes issue #352 (“Corrupt PDF data and ...”), which seemed to intermittently occur when using the method for some problematic PDF files.
- **Fixed** a bug which, in matrix division using the syntax “`m1/m2`”, caused matrix “`m1`” to be **replaced** by the result instead of delivering a new matrix.
- **Fixed** issue #354 (“SyntaxWarning with Python 3.8”). We now always use “`==`” for literals (instead of the “`is`” Python keyword).
- **Fixed** issue #353 (“mupdf version check”), to no longer refuse the import when there are only patch level deviations from MuPDF.

16.32 Changes in Version 1.16.0

This major new version of MuPDF comes with several nice new or changed features. Some of them imply programming API changes, however. This is a synopsis of what has changed:

- PDF document encryption and decryption is now **fully supported**. This includes setting **permissions**, **passwords** (user and owner passwords) and the desired encryption method.
- In response to the new encryption features, PyMuPDF returns an integer (ie. a combination of bits) for document permissions, and no longer a dictionary.
- Redirection of MuPDF errors and warnings is now natively supported. PyMuPDF redirects error messages from MuPDF to `sys.stderr` and no longer buffers them. Warnings continue to be buffered and will not be displayed. Functions exist to access and reset the warnings buffer.
- Annotations are now **only supported for PDF**.

- Annotations and widgets (form fields) are now **separate object chains** on a page (although widgets technically still **are** PDF annotations). This means, that you will **never encounter widgets** when using `Page.firstAnnot` or `Annot.next()`. You must use `Page.firstWidget` and `Widget.next()` to access form fields.
- As part of MuPDF's changes regarding widgets, only the following four fonts are supported, when **adding or changing** form fields: **Courier, Helvetica, Times-Roman and ZapfDingBats**.

List of change details:

- **Added** `Document.can_save_incrementally()` which checks conditions that are preventing use of option `incremental=True` of `Document.save()`.
- **Added** `Page.firstWidget` which points to the first field on a page.
- **Added** `Page.getImageBbox()` which returns the rectangle occupied by an image shown on the page.
- **Added** `Annot.setName()` which lets you change the (icon) name field.
- **Added** outputting the text color in `Page.getText()`: the “dict”, “rawdict” and “xml” options now also show the color in sRGB format.
- **Changed** `Document.permissions` to now contain an integer of bool indicators – was a dictionary before.
- **Changed** `Document.save()`, `Document.write()`, which now fully support password-based decryption and encryption of PDF files.
- **Changed the names of all Python constants** related to annotations and widgets. Please make sure to consult the **Constants and Enumerations** chapter if your script is dealing with these two classes. This decision goes back to the dropped support for non-PDF annotations. The **old names** (starting with “ANNOT_*” or “WIDGET_”) will be available as deprecated synonyms.
- **Changed** font support for widgets: only *Cour* (Courier), *Helv* (Helvetica, default), *TiRo* (Times-Roman) and *ZaDb* (ZapfDingBats) are accepted when **adding or changing** form fields. Only the plain versions are possible – not their italic or bold variations. **Reading** widgets, however will show its original font.
- **Changed** the name of the warnings buffer to `Tools.mupdf_warnings()` and the function to empty this buffer is now called `Tools.reset_mupdf_warnings()`.
- **Changed** `Page.getPixmap()`, `Document.getPagePixmap()`: a new bool argument `annots` can now be used to **suppress the rendering of annotations** on the page.
- **Changed** `Page.addFileAnnot()` and `Page.addTextAnnot()` to enable setting an icon.
- **Removed** widget-related methods and attributes from the `Annot` object.
- **Removed** `Document` attributes `openErrCode`, `openErrMsg`, and `Tools` attributes / methods `stderr`, `reset_stderr`, `stdout`, and `reset_stdout`.
- **Removed thirdparty zlib** dependency in PyMuPDF: there are now compression functions available in MuPDF. Source installers of PyMuPDF may now omit this extra installation step.

16.33 No version published for MuPDF v1.15.0

16.34 Changes in Version 1.14.20 / 1.14.21

- **Changed** text marker annotations to support multiple rectangles / quadrilaterals. This fixes issue #341 (“Question : How to addhighlight so that a string spread across more than a line is covered by one highlight?”) and similar (#285).

- **Fixed** issue #331 (“Importing PyMuPDF changes warning filtering behaviour globally”).

16.35 Changes in Version 1.14.19

- **Fixed** issue #319 (“InsertText function error when use custom font”).
- **Added** new method `Document.getSigFlags()` which returns information on whether a PDF is signed. Resolves issue #326 (“How to detect signature in a form pdf?”).

16.36 Changes in Version 1.14.17

- **Added** `Document.fullcopyPage()` to make full page copies within a PDF (not just copied references as `Document.copyPage()` does).
- **Changed** `Page.getPixmap()`, `Document.getPagePixmap()` now use `alpha=False` as default.
- **Changed** text extraction: the span dictionary now (again) contains its rectangle under the `bbox` key.
- **Changed** `Document.movePage()` and `Document.copyPage()` to use direct functions instead of wrapping `Document.select()` – similar to `Document.deletePage()` in v1.14.16.

16.37 Changes in Version 1.14.16

- **Changed** `Document` methods around PDF `/EmbeddedFiles` to no longer use MuPDF’s “portfolio” functions. That support will be dropped in MuPDF v1.15 – therefore another solution was required.
- **Changed** `Document.embeddedFileCount()` to be a function (was an attribute).
- **Added** new method `Document.embeddedFileNames()` which returns a list of names of embedded files.
- **Changed** `Document.deletePage()` and `Document.deletePageRange()` to internally no longer use `Document.select()`, but instead use functions to perform the deletion directly. As it has turned out, the `Document.select()` method yields invalid outline trees (tables of content) for very complex PDFs and sophisticated use of annotations.

16.38 Changes in Version 1.14.15

- **Fixed** issues #301 (“Line cap and Line join”), #300 (“How to draw a shape without outlines”) and #298 (“utils.updateRect exception”). These bugs pertain to drawing shapes with PyMuPDF. Drawing shapes without any border is fully supported. Line cap styles and line line join style are now differentiated and support all possible PDF values (0, 1, 2) instead of just being a bool. The previous parameter `roundCap` is deprecated in favor of `lineCap` and `lineJoin` and will be deleted in the next release.
- **Fixed** issue #290 (“Memory Leak with getText(‘rawDICT’)”). This bug caused memory not being (completely) freed after invoking the “dict”, “rawdict” and “json” versions of `Page.getText()`.

16.39 Changes in Version 1.14.14

- **Added** new low-level function `ImageProperties()` to determine a number of characteristics for an image.

- **Added** new low-level function `Document.isStream()`, which checks whether an object is of stream type.
- **Changed** low-level functions `Document._getXrefString()` and `Document._getTrailerString()` now by default return object definitions in a formatted form which makes parsing easy.

16.40 Changes in Version 1.14.13

- **Changed** methods working with binary input: while ever supporting bytes and bytearray objects, they now also accept `io.BytesIO` input, using their `getvalue()` method. This pertains to document creation, embedded files, FileAttachment annotations, pixmap creation and others. Fixes issue #274 (“Segfault when using BytesIO as a stream for insertImage”).
- **Fixed** issue #278 (“Is insertImage(keep_proportion=True) broken?”). Images are now correctly presented when keeping aspect ratio.

16.41 Changes in Version 1.14.12

- **Changed** the draw methods of `Page` and `Shape` to support not only RGB, but also GRAY and CMYK colorspaces. This solves issue #270 (“Is there a way to use CMYK color to draw shapes?”). This change also applies to text insertion methods of `Shape`, resp. `Page`.
- **Fixed** issue #269 (“AttributeError in Document.insertPage()”), which occurred when using `Document.insertPage()` with text insertion.

16.42 Changes in Version 1.14.11

- **Changed** `Page.showPDFpage()` to always position the source rectangle centered in the target. This method now also supports **rotation by arbitrary angles**. The argument `reuse_xref` has been deprecated: prevention of duplicates is now **handled internally**.
- **Changed** `Page.insertImage()` to support rotated display of the image and keeping the aspect ratio. Only rotations by multiples of 90 degrees are supported here.
- **Fixed** issue #265 (“TypeError: insertText() got an unexpected keyword argument ‘idx’”). This issue only occurred when using `Document.insertPage()` with also inserting text.

16.43 Changes in Version 1.14.10

- **Changed** `Page.showPDFpage()` to support rotation of the source rectangle. Fixes #261 (“Cannot rotate insterted pages”).
- **Fixed** a bug in `Page.insertImage()` which prevented insertion of multiple images provided as streams.

16.44 Changes in Version 1.14.9

- **Added** new low-level method `Document._getTrailerString()`, which returns the trailer object of a PDF. This is much like `Document._getXrefString()` except that the PDF trailer has no `/xref` to identify it.

- **Added** new parameters for text insertion methods. You can now set stroke and fill colors of glyphs (text characters) independently, as well as the thickness of the glyph border. A new parameter *render_mode* controls the use of these colors, and whether the text should be visible at all.
- **Fixed** issue #258 (“Copying image streams to new PDF without size increase”): For JPX images embedded in a PDF, *Document.extractImage()* will now return them in their original format. Previously, the MuPDF base library was used, which returns them in PNG format (entailing a massive size increase).
- **Fixed** issue #259 (“Morphing text to fit inside rect”). Clarified use of *getTextLength()* and removed extra line breaks for long words.

16.45 Changes in Version 1.14.8

- **Added** *Pixmap.setRect()* to change the pixel values in a rectangle. This is also an alternative to setting the color of a complete pixmap (*Pixmap.clearWith()*).
- **Fixed** an image extraction issue with JBIG2 (monochrome) encoded PDF images. The issue occurred in *Page.getText()* (parameters “dict” and “rawdict”) and in *Document.extractImage()* methods.
- **Fixed** an issue with not correctly clearing a non-alpha *Pixmap* (*Pixmap.clearWith()*).
- **Fixed** an issue with not correctly inverting colors of a non-alpha *Pixmap* (*Pixmap.invertIRect()*).

16.46 Changes in Version 1.14.7

- **Added** *Pixmap.setPixel()* to change one pixel value.
- **Added** documentation for image conversion in the *Collection of Recipes*.
- **Added** new function *getTextLength()* to determine the string length for a given font.
- **Added** Postscript image output (changed *Pixmap.writeImage()* and *Pixmap.getImageData()*).
- **Changed** *Pixmap.writeImage()* and *Pixmap.getImageData()* to ensure valid combinations of color space, alpha and output format.
- **Changed** *Pixmap.writeImage()*: the desired format is now inferred from the filename.
- **Changed** FreeText annotations can now have a transparent background - see *Annot.update()*.

16.47 Changes in Version 1.14.5

- **Changed:** *Shape* methods now strictly use the transformation matrix of the *Page* – instead of “manually” calculating locations.
- **Added** method *Pixmap.pixel()* which returns the pixel value (a list) for given pixel coordinates.
- **Added** method *Pixmap.getImageData()* which returns a bytes object representing the pixmap in a variety of formats. Previously, this could be done for PNG outputs only (*Pixmap.getPNGData()*).
- **Changed:** output of methods *Pixmap.writeImage()* and (the new) *Pixmap.getImageData()* may now also be PSD (Adobe Photoshop Document).
- **Added** method *Shape.drawQuad()* which draws a *Quad*. This actually is a shorthand for a *Shape.drawPolyline()* with the edges of the quad.

- **Changed** method `Shape.drawOval()`: the argument can now be **either** a rectangle (`rect_like`) or a quadrilateral (`quad_like`).

16.48 Changes in Version 1.14.4

- **Fixes** issue #239 “Annotation coordinate consistency”.

16.49 Changes in Version 1.14.3

This patch version contains minor bug fixes and CJK font output support.

- **Added** support for the four CJK fonts as PyMuPDF generated text output. This pertains to methods `Page.insertFont()`, `Shape.insertText()`, `Shape.insertTextbox()`, and corresponding `Page` methods. The new fonts are available under “reserved” fontnames “china-t” (traditional Chinese), “china-s” (simplified Chinese), “japan” (Japanese), and “korea” (Korean).
- **Added** full support for the built-in fonts ‘Symbol’ and ‘Zapfdingbats’.
- **Changed:** The 14 standard fonts can now each be referenced by a 4-letter abbreviation.

16.50 Changes in Version 1.14.1

This patch version contains minor performance improvements.

- **Added** support for `Document` filenames given as `pathlib` object by using the Python `str()` function.

16.51 Changes in Version 1.14.0

To support MuPDF v1.14.0, massive changes were required in PyMuPDF – most of them purely technical, with little visibility to developers. But there are also quite a lot of interesting new and improved features. Following are the details:

- **Added** “ink” annotation.
- **Added** “rubber stamp” annotation.
- **Added** “squiggly” text marker annotation.
- **Added** new class `Quad` (quadrilateral or tetragon) – which represents a general four-sided shape in the plane. The special subtype of rectangular, non-empty tetragons is used in text marker annotations and as returned objects in text search methods.
- **Added** a new option “decrypt” to `Document.save()` and `Document.write()`. Now you can **keep encryption** when saving a password protected PDF.
- **Added** suppression and redirection of unsolicited messages issued by the underlying C-library MuPDF. Consult *Redirecting Error and Warning Messages* for details.
- **Changed:** Changes to annotations now **always require** `Annot.update()` to become effective.
- **Changed** free text annotations to support the full Latin character set and range of appearance options.
- **Changed** text searching, `Page.searchFor()`, to optionally return `Quad` instead `Rect` objects surrounding each search hit.

- **Changed** plain text output: we now add a *n* to each line if it does not itself end with this character.
- **Fixed** issue 211 (“Something wrong in the doc”).
- **Fixed** issue 213 (“Rewritten outline is displayed only by mupdf-based applications”).
- **Fixed** issue 214 (“PDF decryption GONE!”).
- **Fixed** issue 215 (“Formatting of links added with pyMuPDF”).
- **Fixed** issue 217 (“extraction through json is failing for my pdf”).

Behind the curtain, we have changed the implementation of geometry objects: they now purely exist in Python and no longer have “shadow” twins on the C-level (in MuPDF). This has improved processing speed in that area by more than a factor of two.

Because of the same reason, most methods involving geometry parameters now also accept the corresponding Python sequence. For example, in method `page.showPDFpage(rect, ...)` parameter `rect` may now be any *rect_like* sequence.

We also invested considerable effort to further extend and improve the *Collection of Recipes* chapter.

16.52 Changes in Version 1.13.19

This version contains some technical / performance improvements and bug fixes.

- **Changed** memory management: for Python 3 builds, Python memory management is exclusively used across all C-level code (i.e. no more native `malloc()` in MuPDF code or PyMuPDF interface code). This leads to improved memory usage profiles and also some runtime improvements: we have seen > 2% shorter runtimes for text extractions and pixmap creations (on Windows machines only to date).
- **Fixed** an error occurring in Python 2.7, which crashed the interpreter when using `TextPage.extractRAW_DICT()` (= `Page.getText("rawdict")`).
- **Fixed** an error occurring in Python 2.7, when creating link destinations.
- **Extended** the *Collection of Recipes* chapter with more examples.

16.53 Changes in Version 1.13.18

- **Added** method `TextPage.extractRAW_DICT()`, and a corresponding new string parameter “rawdict” to method `Page.getText()`. It extracts text and images from a page in Python *dict* form like `TextPage.extract_DICT()`, but with the detail level of `TextPage.extract_XML()`, which is position information down to each single character.

16.54 Changes in Version 1.13.17

- **Fixed** an error that intermittently caused an exception in `Page.showPDFpage()`, when pages from many different source PDFs were shown.
- **Changed** method `Document.extractImage()` to now return more meta information about the extracted image. Also, its performance has been greatly improved. Several demo scripts have been changed to make use of this method.
- **Changed** method `Document._getXrefStream()` to now return `None` if the object is no stream and no longer raise an exception if otherwise.

- **Added** method `Document._deleteObject()` which deletes a PDF object identified by its *xref*. Only to be used by the experienced PDF expert.
- **Added** a method `PaperRect()` which returns a *Rect* for a supplied paper format string. Example: `fitz.PaperRect("letter") = fitz.Rect(0.0, 0.0, 612.0, 792.0)`.
- **Added** a *Collection of Recipes* chapter to this document.

16.55 Changes in Version 1.13.16

- **Added** support for correctly setting transparency (opacity) for certain annotation types.
- **Added** a tool property (`Tools.fitz_config`) showing the configuration of this PyMuPDF version.
- **Fixed** issue #193 ('insertText(overlay=False) gives "cannot resize a buffer with shared storage" error') by avoiding read-only buffers.

16.56 Changes in Version 1.13.15

- **Fixed** issue #189 ("cannot find builtin CJK font"), so we are supporting builtin CJK fonts now (CJK = China, Japan, Korea). This should lead to correctly generated pixmaps for documents using these languages. This change has consequences for our binary file size: it will now range between 8 and 10 MB, depending on the OS.
- **Fixed** issue #191 ("Jupyter notebook kernel dies after ca. 40 pages"), which occurred when modifying the contents of an annotation.

16.57 Changes in Version 1.13.14

This patch version contains several improvements, mainly for annotations.

- **Changed** `Annot.lineEnds` is now a list of two integers representing the line end symbols. Previously was a *dict* of strings.
- **Added** support of line end symbols for applicable annotations. PyMuPDF now can generate these annotations including the line end symbols.
- **Added** `Annot.setLineEnds()` adds line end symbols to applicable annotation types ('Line', 'PolyLine', 'Polygon').
- **Changed** technical implementation of `Page.insertImage()` and `Page.showPDFpage()`: they now create their own contents objects, thereby avoiding changes of potentially large streams with consequential compression / decompression efforts and high change volumes with incremental updates.

16.58 Changes in Version 1.13.13

This patch version contains several improvements for embedded files and file attachment annotations.

- **Added** `Document.embeddedFileUpd()` which allows changing **file content and metadata** of an embedded file. It supersedes the old method `Document.embeddedFileSetInfo()` (which will be deleted in a future version). Content is automatically compressed and metadata may be unicode.
- **Changed** `Document.embeddedFileAdd()` to now automatically compress file content. Accompanying metadata can now be unicode (had to be ASCII in the past).

- **Changed** `Document.embeddedFileDel()` to now automatically delete **all entries** having the supplied identifying name. The return code is now an integer count of the removed entries (was *None* previously).
- **Changed** embedded file methods to now also accept or show the PDF unicode filename as additional parameter *ufilename*.
- **Added** `Page.addFileAnnot()` which adds a new file attachment annotation.
- **Changed** `Annot.fileUpd()` (file attachment annot) to now also accept the PDF unicode *ufilename* parameter. The description parameter *desc* correctly works with unicode. Furthermore, **all** parameters are optional, so metadata may be changed without also replacing the file content.
- **Changed** `Annot.fileInfo()` (file attachment annot) to now also show the PDF unicode filename as parameter *ufilename*.
- **Fixed** issue #180 (“page.getText(output='dict') return invalid bbox”) to now also work for vertical text.
- **Fixed** issue #185 (“Can’t render the annotations created by PyMuPDF”). The issue’s cause was the minimalistic MuPDF approach when creating annotations. Several annotation types have no */AP* (“appearance”) object when created by MuPDF functions. MuPDF, SumatraPDF and hence also PyMuPDF cannot render annotations without such an object. This fix now ensures, that an appearance object is always created together with the annotation itself. We still do not support line end styles.

16.59 Changes in Version 1.13.12

- **Fixed** issue #180 (“page.getText(output='dict') return invalid bbox”). Note that this is a circumvention of an MuPDF error, which generates zero-height character rectangles in some cases. When this happens, this fix ensures a bbox height of at least fontsize.
- **Changed** for `ListBox` and `ComboBox` widgets, the attribute list of selectable values has been renamed to `Widget.choice_values`.
- **Changed** when adding widgets, any missing of the *PDF Base 14 Fonts* is automatically added to the PDF. Widget text fonts can now also be chosen from existing widget fonts. Any specified field values are now honored and lead to a field with a preset value.
- **Added** `Annot.updateWidget()` which allows changing existing form fields – including the field value.

16.60 Changes in Version 1.13.11

While the preceding patch subversions only contained various fixes, this version again introduces major new features:

- **Added** basic support for PDF widget annotations. You can now add PDF form fields of types `Text`, `CheckBox`, `ListBox` and `ComboBox`. Where necessary, the PDF is transformed to a Form PDF with the first added widget.
- **Fixed** issues #176 (“wrong file embedding”), #177 (“segment fault when invoking page.getText()”) and #179 (“Segmentation fault using page.getLinks() on encrypted PDF”).

16.61 Changes in Version 1.13.7

- **Added** support of variable page sizes for reflowable documents (e-books, HTML, etc.): new parameters *rect* and *fontsize* in `Document` creation (open), and as a separate method `Document.layout()`.
- **Added** `Annot` creation of many annotations types: sticky notes, free text, circle, rectangle, line, polygon, poly-line and text markers.

- **Added** support of annotation transparency (`Annot.opacity`, `Annot.setOpacity()`).
- **Changed** `Annot.vertices`: point coordinates are now grouped as pairs of floats (no longer as separate floats).
- **Changed** annotation colors dictionary: the two keys are now named “*stroke*” (formerly “*common*”) and “*fill*”.
- **Added** `Document.isDirty` which is *True* if a PDF has been changed in this session. Reset to *False* on each `Document.save()` or `Document.write()`.

16.62 Changes in Version 1.13.6

- Fix #173: for memory-resident documents, ensure the stream object will not be garbage-collected by Python before document is closed.

16.63 Changes in Version 1.13.5

- New low-level method `Page._setContentts()` defines an object given by its *xref* to serve as the *contents* object.
- Changed and extended PDF form field support: the attribute `widget_text` has been renamed to `Annot.widget_value`. Values of all form field types (except signatures) are now supported. A new attribute `Annot.widget_choices` contains the selectable values of listboxes and comboboxes. All these attributes now contain *None* if no value is present.

16.64 Changes in Version 1.13.4

- `Document.convertToPDF()` now supports page ranges, reverted page sequences and page rotation. If the document already is a PDF, an exception is raised.
- Fixed a bug (introduced with v1.13.0) that prevented `Page.insertImage()` for transparent images.

16.65 Changes in Version 1.13.3

Introduces a way to convert **any MuPDF supported document** to a PDF. If you ever wanted PDF versions of your XPS, EPUB, CBZ or FB2 files – here is a way to do this.

- `Document.convertToPDF()` returns a Python *bytes* object in PDF format. Can be opened like normal in PyMuPDF, or be written to disk with the “.pdf” extension.

16.66 Changes in Version 1.13.2

The major enhancement is PDF form field support. Form fields are annotations of type (19, ‘*Widget*’). There is a new document method to check whether a PDF is a form. The *Annot* class has new properties describing field details.

- `Document.isFormPDF` is true if object type */AcroForm* and at least one form field exists.
- `Annot.widget_type`, `Annot.widget_text` and `Annot.widget_name` contain the details of a form field (i.e. a “Widget” annotation).

16.67 Changes in Version 1.13.1

- `TextPage.extractDICT()` is a new method to extract the contents of a document page (text and images). All document types are supported as with the other `TextPage.extract*()` methods. The returned object is a dictionary of nested lists and other dictionaries, and **exactly equal** to the JSON-deserialization of the old `TextPage.extractJSON()`. The difference is that the result is created directly – no JSON module is used. Because the user needs no JSON module to interpret the information, it should be easier to use, and also have a better performance, because it contains images in their original **binary format** – they need not be base64-decoded.
- `Page.getText()` correspondingly supports the new parameter value “*dict*” to invoke the above method.
- `TextPage.extractJSON()` (resp. `Page.getText(“json”)`) is still supported for convenience, but its use is expected to decline.

16.68 Changes in Version 1.13.0

This version is based on MuPDF v1.13.0. This release is “primarily a bug fix release”.

In PyMuPDF, we are also doing some bug fixes while introducing minor enhancements. There only very minimal changes to the user’s API.

- `Document` construction is more flexible: the new `filetype` parameter allows setting the document type. If specified, any extension in the filename will be ignored. More completely addresses [issue #156](#). As part of this, the documentation has been reworked.
- **Changes to `Pixmap` constructors:**
 - Colorspace conversion no longer allows dropping the alpha channel: source and target **alpha will now always be the same**. We have seen exceptions and even interpreter crashes when using `alpha = 0`.
 - As a replacement, the simple `pixmap` copy lets you choose the target alpha.
- `Document.save()` again offers the full garbage collection range 0 thru 4. Because of a bug in `xref` maintenance, we had to temporarily enforce `garbage > 1`. Finally resolves [issue #148](#).
- `Document.save()` now offers to “prettify” PDF source via an additional argument.
- `Page.insertImage()` has the additional `stream`-parameter, specifying a memory area holding an image.
- Issue with garbled PNGs on Linux systems has been resolved (“[Problem writing PNG](#)” #133).

16.69 Changes in Version 1.12.4

This is an extension of 1.12.3.

- Fix of [issue #147](#): methods `Document.getPageFontlist()` and `Document.getPageImagelist()` now also show fonts and images contained in `resources` nested via “Form XObjects”.
- Temporary fix of [issue #148](#): Saving to new PDF files will now automatically use `garbage = 2` if a lower value is given. Final fix is to be expected with MuPDF’s next version. At that point we will remove this circumvention.
- Preventive fix of illegally using stencil / image mask `pixmap`s in some methods.
- Method `Document.getPageFontlist()` now includes the encoding name for each font in the list.

- Method `Document.getPageImagelist()` now includes the decode method name for each image in the list.

16.70 Changes in Version 1.12.3

This is an extension of 1.12.2.

- Many functions now return `None` instead of `0`, if the result has no other meaning than just indicating successful execution (`Document.close()`, `Document.save()`, `Document.select()`, `Pixmap.writePNG()` and many others).

16.71 Changes in Version 1.12.2

This is an extension of 1.12.1.

- Method `Page.showPDFpage()` now accepts the new `clip` argument. This specifies an area of the source page to which the display should be restricted.
- New `Page.CropBox` and `Page.MediaBox` have been included for convenience.

16.72 Changes in Version 1.12.1

This is an extension of version 1.12.0.

- New method `Page.showPDFpage()` displays another's PDF page. This is a **vector** image and therefore remains precise across zooming. Both involved documents must be PDF.
- New method `Page.getSVGimage()` creates an SVG image from the page. In contrast to the raster image of a pixmap, this is a vector image format. The return is a unicode text string, which can be saved in a `.svg` file.
- Method `Page.getTextBlocks()` now accepts an additional bool parameter "images". If set to true (default is false), image blocks (metadata only) are included in the produced list and thus allow detecting areas with rendered images.
- Minor bug fixes.
- "text" result of `Page.getText()` concatenates all lines within a block using a single space character. MuPDF's original uses "\n" instead, producing a rather ragged output.
- New properties of `Page` objects `Page.MediaBoxSize` and `Page.CropBoxPosition` provide more information about a page's dimensions. For non-PDF files (and for most PDF files, too) these will be equal to `Page.rect.bottom_right`, resp. `Page.rect.top_left`. For example, class `Shape` makes use of them to correctly position its items.

16.73 Changes in Version 1.12.0

This version is based on and requires MuPDF v1.12.0. The new MuPDF version contains quite a number of changes – most of them around text extraction. Some of the changes impact the programmer's API.

- `Outline.saveText()` and `Outline.saveXML()` have been deleted without replacement. You probably haven't used them much anyway. But if you are looking for a replacement: the output of `Document.getToC()` can easily be used to produce something equivalent.

- Class *TextSheet* does no longer exist.
- Text “spans” (one of the hierarchy levels of *TextPage*) no longer contain positioning information (i.e. no “bbox” key). Instead, spans now provide the font information for its text. This impacts our JSON output variant.
- HTML output has improved very much: it now creates valid documents which can be displayed by browsers to produce a similar view as the original document.
- There is a new output format XHTML, which provides text and images in a browser-readable format. The difference to HTML output is, that no effort is made to reproduce the original layout.
- All output formats of *Page.getText()* now support creating complete, valid documents, by wrapping them with appropriate header and trailer information. If you are interested in using the HTML output, please make sure to read *Controlling Quality of HTML Output*.
- To support finding text positions, we have added special methods that don’t need detours like *TextPage.extractJSON()* or *TextPage.extractXML()*: use *Page.getTextBlocks()* or resp. *Page.getTextWords()* to create lists of text blocks or resp. words, which are accompanied by their rectangles. This should be much faster than the standard text extraction methods and also avoids using additional packages for interpreting their output.

16.74 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.

16.75 Changes in Version 1.11.1

This is an extension of v1.11.0.

- New class *Shape*. It facilitates and extends the creation of image shapes on PDF pages. It contains multiple methods for creating elementary shapes like lines, rectangles or circles, which can be combined into more complex ones and be given common properties like line width or colors. Combined shapes are handled as a unit and e.g. be “morphed” together. The class can accumulate multiple complex shapes and put them all in the page’s foreground or background – thus also reducing the number of updates to the page’s *contents* object.
- All *Page* draw methods now use the new *Shape* class.
- Text insertion methods *insertText()* and *insertTextBox()* now support morphing in addition to text rotation. They have become part of the *Shape* class and thus allow text to be freely combined with graphics.
- A new *Pixmap* constructor allows creating pixmap copies with an added alpha channel. A new method also allows directly manipulating alpha values.

- Binary algebraic operations with geometry objects (matrices, rectangles and points) now generally also support lists or tuples as the second operand. You can add a tuple (x, y) of numbers to a *Point*. In this context, such sequences are called “*point_like*” (resp. *matrix_like*, *rect_like*).
- Geometry objects now fully support in-place operators. For example, $p /= m$ replaces point p with $p * 1/m$ for a number, or $p * \sim m$ 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)*.

16.76 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 and 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.

16.77 Changes in Version 1.10.0

16.77.1 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 compatibility to existing code.
- In an effort to minimize memory requirements, several improvements have been built into MuPDF v1.10:
 - A new *config.h* file can be used to de-select unwanted features in the C base code. Using this feature we have been able to reduce the size of our binary *_fitz.o / _fitz.pyd* by about 50% (from 9 MB to 4.5 MB). When UPX-ing this, the size goes even further down to a very handy 2.3 MB.
 - The alpha (transparency) channel for pixmaps is now optional. Letting alpha default to *False* significantly reduces pixmap sizes (by 20% – CMYK, 25% – RGB, 50% – GRAY). Many *Pixmap* constructors therefore now accept an *alpha* boolean to control inclusion of this channel. Other pixmap constructors (e.g. those for file and image input) create pixmaps with no alpha altogether. On the downside, save methods for pixmaps no longer accept a *savealpha* option: this channel will always be saved when present. To minimize code breaks, we have left this parameter in the call patterns – it will just be ignored.
- *DisplayList* and *TextPage* class constructors now **require the mediabox** of the page they are referring to (i.e. the *page.bound()* rectangle). There is no way to construct this information from other sources, therefore a source code change cannot be avoided in these cases. We assume however, that not many users are actually employing these rather low level classes explicitly. So the impact of that change should be minor.

16.77.2 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.

16.78 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.

16.79 Changes in Version 1.9.2

This version is also based on MuPDF v1.9a. Changes compared to version 1.9.1:

- *fitz.open()* (no parameters) creates a new empty **PDF** document, i.e. if saved afterwards, it must be given a *.pdf* extension.
- *Document* now accepts all of the following formats (*Document* and *open* are synonyms):
 - *open()*,
 - *open(filename)* (equivalent to *open(filename, None)*),
 - *open filetype, area* (equivalent to *open filetype, stream = area*).

Type of memory area *stream* may be *bytes* or *bytearray*. Thus, e.g. *area = open("file.pdf", "rb").read()* may be used directly (without first converting it to *bytearray*).

- New method *Document.insertPDF()* (PDFs only) inserts a range of pages from another PDF.
- *Document* objects *doc* now support the *len()* function: *len(doc) == doc.pageCount*.
- New method *Document.getPageImageList()* creates a list of images used on a page.
- New method *Document.getPageFontList()* creates a list of fonts referenced by a page.
- New pixmap constructor *fitz.Pixmap(doc, xref)* creates a pixmap based on an opened PDF document and an *xref* number of the image.
- New pixmap constructor *fitz.Pixmap(cspace, spix)* creates a pixmap as a copy of another one *spix* with the colorspace converted to *cspace*. This works for all colorspace combinations.
- Pixmap constructor *fitz.Pixmap(colorspace, width, height, samples)* now allows *samples* to also be *bytes*, not only *bytearray*.

16.80 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.

Symbols

__init__() (*Colorspace method*), 97
 __init__() (*Device method*), 254
 __init__() (*DisplayList method*), 98
 __init__() (*Document method*), 101
 __init__() (*IRect method*), 136
 __init__() (*Matrix method*), 143
 __init__() (*Pixmap method*), 180–182
 __init__() (*Point method*), 188
 __init__() (*Quad method*), 191
 __init__() (*Rect method*), 194
 __init__() (*Shape method*), 198
 __init__() (*TextWriter method*), 221
 _getContents() (*Page method*), 248
 _getOLRootNumber() (*Document method*), 252
 _getPageObjNumber() (*Document method*), 247
 _make_page_map() (*Document method*), 247
 _setContents() (*Page method*), 249

A

a (*Matrix attribute*), 145
 abs_unit (*Point attribute*), 190
 add_layer_config() (*Document method*), 103
 add_ocg() (*Document method*), 104
 addCaretAnnot() (*Page method*), 154
 addCircleAnnot() (*Page method*), 156
 addFileAnnot
 examples, 20
 addFileAnnot() (*Page method*), 155
 addFreetextAnnot
 align, 155
 color, 155
 fontname, 155
 fontsize, 155
 rect, 155
 rotate, 155
 addFreetextAnnot() (*Page method*), 155
 addHighlightAnnot() (*Page method*), 158
 addInkAnnot() (*Page method*), 156

addLineAnnot() (*Page method*), 156
 addPolygonAnnot() (*Page method*), 158
 addPolylineAnnot() (*Page method*), 158
 addRectAnnot() (*Page method*), 156
 addRedactAnnot() (*Page method*), 156
 addSquigglyAnnot() (*Page method*), 158
 addStampAnnot() (*Page method*), 159
 addStrikeoutAnnot() (*Page method*), 158
 addTextAnnot() (*Page method*), 154
 addUnderlineAnnot() (*Page method*), 158
 addWidget() (*Page method*), 160
 adobe_glyph_names(), 242
 adobe_glyph_unicodes(), 242
 align
 addFreetextAnnot, 155
 insertTextbox, 162, 204
 alpha
 get_pixmap, 86
 getPixmap, 98, 170
 alpha (*Pixmap attribute*), 185
 Annot (*built-in class*), 86
 Annot.get_text
 blocks, 87
 clip, 87
 dict, 87
 flags, 87
 html, 87
 json, 87
 rawdict, 87
 text, 87
 words, 87
 xhtml, 87
 xml, 87
 annot_names() (*Page method*), 171
 annot_xrefs() (*Page method*), 171
 annots
 getPixmap, 170
 insertPDF (*Document method*), 119
 annots() (*Page method*), 161
 append() (*TextWriter method*), 222

`appendv()` (*TextWriter method*), 222
`apply_redactions()` (*Page method*), 160
`ascender` (*Font attribute*), 135
`attach`
 embed file, 57
`authenticate()` (*Document method*), 108

B

`b` (*Matrix attribute*), 145
`Base14_Fonts` (*built-in variable*), 261
`bbox` (*Font attribute*), 135
`b1` (*IRect attribute*), 138
`b1` (*Rect attribute*), 196
`blend_mode`
 update, 91
`blendmode` (*Annot attribute*), 89
`blocks`
 Annot.get_text, 87
 getText, 167
`border` (*Annot attribute*), 95
`border` (*Link attribute*), 140
`border_color`
 update, 91
`border_color` (*Widget attribute*), 229
`border_dashes` (*Widget attribute*), 229
`border_style` (*Widget attribute*), 229
`border_width`
 insertText, 162, 203
 insertTextbox, 162, 204
`border_width` (*Widget attribute*), 229
`bottom_left` (*IRect attribute*), 138
`bottom_left` (*Rect attribute*), 196
`bottom_right` (*IRect attribute*), 138
`bottom_right` (*Rect attribute*), 197
`bound()` (*Page method*), 154
`br` (*IRect attribute*), 138
`br` (*Rect attribute*), 197
`breadth`
 drawSquiggle, 163, 199
 drawZigzag, 163, 200
`buffer`
 update_file, 92
`buffer` (*Font attribute*), 134
`button_caption` (*Widget attribute*), 230

C

`c` (*Matrix attribute*), 145
`can_save_incrementally()` (*Document method*), 117
`catalog` (*built-in variable*), 257
`chapterCount` (*Document attribute*), 127
`chapterPageCount()` (*Document method*), 108
`choice_values` (*Widget attribute*), 230
`clean_contents()` (*Annot method*), 250

`clean_contents()` (*Page method*), 249
`clearWith()` (*Pixmap method*), 182
`clip`
 Annot.get_text, 87
 getPixmap, 98, 170
 getText, 167
 showPDFpage, 172
`close()` (*Document method*), 124
`closePath`
 drawBezier, 163
 drawCircle, 163
 drawCurve, 163
 drawLine, 163
 drawOval, 163
 drawPolyline, 163
 drawRect, 164
 drawSector, 163
 drawSquiggle, 163
 drawZigzag, 163
 finish, 206
`color`
 addFreetextAnnot, 155
 drawBezier, 163
 drawCircle, 163
 drawCurve, 163
 drawLine, 163
 drawOval, 163
 drawPolyline, 163
 drawRect, 164
 drawSector, 163
 drawSquiggle, 163
 drawZigzag, 163
 finish, 206
 insertPage (*Document method*), 120
 insertText, 162, 203
 insertTextbox, 162, 204
`color` (*TextWriter attribute*), 223
`colors` (*Annot attribute*), 94
`colors` (*Link attribute*), 140
`colorspace`
 get_pixmap, 86
 getPixmap, 98, 170
`Colorspace` (*built-in class*), 97
`colorspace` (*Pixmap attribute*), 185
`commit`
 overlay, 207
`commit()` (*Shape method*), 207
`concat()` (*Matrix method*), 144
`contains()` (*IRect method*), 137
`contains()` (*Rect method*), 196
`contents` (*built-in variable*), 257
`ConversionHeader()`, 245
`ConversionTrailer()`, 246
`convertToPDF`

examples, 17
 convertToPDF (*Document method*), 111
 from_page, 111
 rotate, 111
 to_page, 111
 convertToPDF () (*Document method*), 111
 copyPage () (*Document method*), 121
 copyPixmap
 examples, 24, 25
 copyPixmap () (*Pixmap method*), 184
 CropBox (*built-in variable*), 257
 CropBox (*Page attribute*), 176
 CropBoxPosition (*Page attribute*), 176
 cross_out
 update, 91
 CS_CMYK (*built-in variable*), 261
 CS_GRAY (*built-in variable*), 261
 CS_RGB (*built-in variable*), 261
 csCMYK (*built-in variable*), 261
 csGRAY (*built-in variable*), 261
 csRGB (*built-in variable*), 261

D

d (*Matrix attribute*), 145
 dashes
 drawBezier, 163
 drawCircle, 163
 drawCurve, 163
 drawLine, 163
 drawOval, 163
 drawPolyline, 163
 drawRect, 164
 drawSector, 163
 drawSquiggle, 163
 drawZigzag, 163
 finish, 206
 del_toc_item () (*Document method*), 116
 del_xml_metadata () (*Document method*), 246
 delete
 pages, 58
 delete_object () (*Document method*), 246
 delete_responses () (*Annot method*), 91
 deleteAnnot () (*Page method*), 160
 deleteLink () (*Page method*), 161
 deletePage () (*Document method*), 120
 deletePageRange () (*Document method*), 121
 deleteWidget () (*Page method*), 160
 derotationMatrix (*Page attribute*), 176
 desc
 embeddedFileAdd (*Document method*), 122
 embeddedFileUpd (*Document method*), 124
 update_file, 92
 descender (*Font attribute*), 135
 dest (*Link attribute*), 141

dest (*linkDest attribute*), 141
 dest (*Outline attribute*), 151
 Device (*built-in class*), 254
 dict
 Annot.get_text, 87
 getText, 167
 dictionary (*built-in variable*), 258
 DisplayList (*built-in class*), 98
 distance_to () (*Point method*), 189
 doc (*Shape attribute*), 208
 Document
 filename, 101
 filetype, 101
 fontsize, 101
 open, 101
 rect, 101
 stream, 101
 Document (*built-in class*), 101
 down (*Outline attribute*), 151
 draw_cont (*Shape attribute*), 208
 drawBezier
 closePath, 163
 color, 163
 dashes, 163
 fill, 163
 fill_opacity, 163
 lineCap, 163
 lineJoin, 163
 morph, 163
 oc, 163
 overlay, 163
 stroke_opacity, 163
 width, 163
 drawBezier () (*Page method*), 163
 drawBezier () (*Shape method*), 200
 drawCircle
 closePath, 163
 color, 163
 dashes, 163
 fill, 163
 fill_opacity, 163
 lineCap, 163
 lineJoin, 163
 morph, 163
 oc, 163
 overlay, 163
 stroke_opacity, 163
 width, 163
 drawCircle () (*Page method*), 163
 drawCircle () (*Shape method*), 201
 drawCurve
 closePath, 163
 color, 163
 dashes, 163

- fill, 163
- fill_opacity, 163
- lineCap, 163
- lineJoin, 163
- morph, 163
- oc, 163
- overlay, 163
- stroke_opacity, 163
- width, 163
- drawCurve() (*Page method*), 163
- drawCurve() (*Shape method*), 202
- drawLine
 - closePath, 163
 - color, 163
 - dashes, 163
 - fill, 163
 - fill_opacity, 163
 - lineCap, 163
 - lineJoin, 163
 - morph, 163
 - oc, 163
 - overlay, 163
 - stroke_opacity, 163
 - width, 163
- drawLine() (*Page method*), 163
- drawLine() (*Shape method*), 198
- drawOval
 - closePath, 163
 - color, 163
 - dashes, 163
 - fill, 163
 - fill_opacity, 163
 - lineCap, 163
 - lineJoin, 163
 - morph, 163
 - oc, 163
 - overlay, 163
 - stroke_opacity, 163
 - width, 163
- drawOval() (*Page method*), 163
- drawOval() (*Shape method*), 201
- drawPolyline
 - closePath, 163
 - color, 163
 - dashes, 163
 - fill, 163
 - fill_opacity, 163
 - lineCap, 163
 - lineJoin, 163
 - morph, 163
 - oc, 163
 - overlay, 163
 - stroke_opacity, 163
 - width, 163
- drawPolyline() (*Page method*), 163
- drawPolyline() (*Shape method*), 200
- drawQuad() (*Shape method*), 203
- drawRect
 - closePath, 164
 - color, 164
 - dashes, 164
 - fill, 164
 - fill_opacity, 164
 - lineCap, 164
 - lineJoin, 164
 - morph, 164
 - oc, 164
 - overlay, 164
 - stroke_opacity, 164
 - width, 164
- drawRect() (*Page method*), 164
- drawRect() (*Shape method*), 203
- drawSector
 - closePath, 163
 - color, 163
 - dashes, 163
 - fill, 163
 - fill_opacity, 163
 - fullSector, 163, 202
 - lineCap, 163
 - lineJoin, 163
 - morph, 163
 - oc, 163
 - overlay, 163
 - stroke_opacity, 163
 - width, 163
- drawSector() (*Page method*), 163
- drawSector() (*Shape method*), 202
- drawSquiggle
 - breadth, 163, 199
 - closePath, 163
 - color, 163
 - dashes, 163
 - fill, 163
 - fill_opacity, 163
 - lineCap, 163
 - lineJoin, 163
 - morph, 163
 - oc, 163
 - overlay, 163
 - stroke_opacity, 163
 - width, 163
- drawSquiggle() (*Page method*), 163
- drawSquiggle() (*Shape method*), 199
- drawZigzag
 - breadth, 163, 200
 - closePath, 163
 - color, 163

- dashes, 163
 - fill, 163
 - fill_opacity, 163
 - lineCap, 163
 - lineJoin, 163
 - morph, 163
 - oc, 163
 - overlay, 163
 - stroke_opacity, 163
 - width, 163
 - drawZigzag() (*Page method*), 163
 - drawZigzag() (*Shape method*), 200
- ## E
- e (*Matrix attribute*), 145
 - embed
 - file, attach, 57
 - PDF, picture, 20
 - embeddedFileAdd
 - examples, 20, 22
 - embeddedFileAdd (*Document method*)
 - desc, 122
 - filename, 122
 - ufilename, 122
 - embeddedFileAdd() (*Document method*), 122
 - embeddedFileCount() (*Document method*), 123
 - embeddedFileDel() (*Document method*), 123
 - embeddedFileGet() (*Document method*), 123
 - embeddedFileInfo() (*Document method*), 123
 - embeddedFileNames() (*Document method*), 124
 - embeddedFileSetInfo() (*Document method*), 124
 - embeddedFileUpd (*Document method*)
 - desc, 124
 - filename, 124
 - ufilename, 124
 - embeddedFileUpd() (*Document method*), 124
 - encoding
 - insertFont, 164
 - insertText, 162, 203
 - insertTextbox, 162, 204
 - even_odd
 - finish, 206
 - examples
 - addFileAnnot, 20
 - convertToPDF, 17
 - copyPixmap, 24, 25
 - embeddedFileAdd, 20, 22
 - extractImage, 17
 - getImageData, 22
 - insertImage, 20, 22
 - invertIRect, 25
 - JPEG, 22
 - PhotoImage, 22
 - Photoshop, 22
 - Postscript, 22
 - setRect, 25
 - showPDFpage, 20, 22
 - writeImage, 22, 25
 - expandtabs
 - insertTextbox, 162, 204
 - extract
 - image non-PDF, 17
 - image PDF, 17
 - table, 31
 - text rectangle, 29
 - extractBLOCKS() (*TextPage method*), 214
 - extractDICT() (*TextPage method*), 215
 - extractFont() (*Document method*), 253
 - extractHTML() (*TextPage method*), 214
 - extractImage
 - examples, 17
 - extractImage() (*Document method*), 252
 - extractJSON() (*TextPage method*), 215
 - extractRAWDICT() (*TextPage method*), 215
 - extractRAWJSON() (*TextPage method*), 215
 - extractTEXT() (*TextPage method*), 214
 - extractText() (*TextPage method*), 214
 - extractWORDS() (*TextPage method*), 214
 - extractXHTML() (*TextPage method*), 215
 - extractXML() (*TextPage method*), 215
- ## F
- f (*Matrix attribute*), 145
 - field_flags (*Widget attribute*), 230
 - field_label (*Widget attribute*), 230
 - field_name (*Widget attribute*), 230
 - field_type (*Widget attribute*), 230
 - field_type_string (*Widget attribute*), 230
 - field_value (*Widget attribute*), 230
 - file
 - attach embed, 57
 - file extension
 - wrong, 57
 - file_info() (*Annot method*), 92
 - filename
 - Document, 101
 - embeddedFileAdd (*Document method*), 122
 - embeddedFileUpd (*Document method*), 124
 - insertImage, 165
 - open, 101
 - update_file, 92
 - fileSpec (*linkDest attribute*), 141
 - filetype
 - Document, 101
 - open, 101
 - fill
 - drawBezier, 163
 - drawCircle, 163

- drawCurve, 163
- drawLine, 163
- drawOval, 163
- drawPolyline, 163
- drawRect, 164
- drawSector, 163
- drawSquiggle, 163
- drawZigzag, 163
- finish, 206
- insertText, 162, 203
- insertTextbox, 162, 204
- fill_color
 - update, 91
- fill_color (*Widget attribute*), 230
- fill_opacity
 - drawBezier, 163
 - drawCircle, 163
 - drawCurve, 163
 - drawLine, 163
 - drawOval, 163
 - drawPolyline, 163
 - drawRect, 164
 - drawSector, 163
 - drawSquiggle, 163
 - drawZigzag, 163
 - finish, 206
 - insertText, 162, 203
 - insertTextbox, 162
- fillTextbox() (*TextWriter method*), 222
- findBookmark() (*Document method*), 108
- finish
 - closePath, 206
 - color, 206
 - dashes, 206
 - even_odd, 206
 - fill, 206
 - fill_opacity, 206
 - lineCap, 206
 - lineJoin, 206
 - morph, 206
 - oc, 206
 - stroke_opacity, 206
 - width, 206
- finish() (*Shape method*), 206
- firstAnnot (*Page attribute*), 177
- firstLink (*Page attribute*), 177
- firstWidget (*Page attribute*), 177
- fitz_config (*Tools attribute*), 227
- fitz_fontdescriptors, 244
- flags
 - Annot.get_text, 87
 - getText, 167
 - getTextPage, 168
 - searchFor, 174
 - flags (*Annot attribute*), 94
 - flags (*Font attribute*), 135
 - flags (*linkDest attribute*), 141
- Font (*built-in class*), 130
- fontbuffer
 - insertFont, 164
- fontfile
 - insertFont, 164
 - insertPage (*Document method*), 120
 - insertText, 162, 203
 - insertTextbox, 162, 204
- FontInfos (*Document attribute*), 254
- fontname
 - addFreetextAnnot, 155
 - insertFont, 164
 - insertPage (*Document method*), 120
 - insertText, 162, 203
 - insertTextbox, 162, 204
- fontsize
 - addFreetextAnnot, 155
 - Document, 101
 - insertPage (*Document method*), 120
 - insertText, 162, 203
 - insertTextbox, 162, 204
 - layout (*Document method*), 114
 - open, 101
 - update, 91
- FormFonts (*Document attribute*), 128
- from_page
 - convertToPDF (*Document method*), 111
 - insertPDF (*Document method*), 119
- fullcopyPage() (*Document method*), 121
- fullSector
 - drawSector, 163, 202

G

- gammaWith() (*Pixmap method*), 182
- gen_id() (*Tools method*), 224
- get_file() (*Annot method*), 92
- get_new_xref() (*Document method*), 251
- get_oc() (*Annot method*), 88
- get_oc() (*Document method*), 102
- get_oc_states() (*Document method*), 105
- get_ocgs() (*Document method*), 106
- get_ocmd() (*Document method*), 105
- get_pixmap
 - alpha, 86
 - colorspace, 86
 - matrix, 86
- get_pixmap() (*Annot method*), 86
- get_sound() (*Annot method*), 92
- get_text() (*Annot method*), 87
- get_textbox() (*Annot method*), 87
- getArea() (*IRect method*), 137

- getArea () (*Rect method*), 195
 - getCharWidths () (*Document method*), 250
 - getDisplayList () (*Page method*), 248
 - getDrawings () (*Page method*), 168
 - getFontList () (*Page method*), 169
 - getImageBbox () (*Page method*), 169
 - getImageData
 - examples, 22
 - getImageData () (*Pixmap method*), 185
 - getImageList () (*Page method*), 169
 - getLinks () (*Page method*), 161
 - getPageFontList () (*Document method*), 113
 - getPageImageList () (*Document method*), 113
 - getPagePixmap () (*Document method*), 112
 - getPageText () (*Document method*), 114
 - getPageXObjectList () (*Document method*), 112
 - getPDFnow (), 244
 - getPDFstr (), 245
 - getPixmap
 - alpha, 98, 170
 - annots, 170
 - clip, 98, 170
 - colorspace, 98, 170
 - matrix, 98, 170
 - getPixmap () (*DisplayList method*), 98
 - getPixmap () (*Page method*), 170
 - getPNGData () (*Pixmap method*), 185
 - getPNGdata () (*Pixmap method*), 185
 - getRect () (*IRect method*), 137
 - getRectArea () (*IRect method*), 137
 - getRectArea () (*Rect method*), 195
 - getSigFlags () (*Document method*), 122
 - getSVGimage
 - matrix, 170
 - getSVGimage () (*Page method*), 170
 - getText
 - blocks, 167
 - clip, 167
 - dict, 167
 - flags, 167
 - html, 167
 - json, 167
 - rawdict, 167
 - text, 167
 - words, 167
 - xhtml, 167
 - xml, 167
 - getText () (*Page method*), 167
 - getTextBlocks () (*Page method*), 248
 - getTextbox () (*Page method*), 168
 - getTextlength (), 244
 - getTextPage
 - flags, 168
 - getTextPage () (*DisplayList method*), 99
 - getTextPage () (*Page method*), 168
 - getTextWords () (*Page method*), 248
 - getTOC () (*Document method*), 112
 - getToC () (*Document method*), 112
 - getXmlMetadata () (*Document method*), 115
 - glyph_advance () (*Font method*), 133
 - glyph_bbox () (*Font method*), 134
 - glyph_count (*Font attribute*), 135
 - glyph_name_to_unicode (), 241
 - glyph_name_to_unicode () (*Font method*), 134
- ## H
- h (*Pixmap attribute*), 186
 - has_glyph () (*Font method*), 132
 - has_popup (*Annot attribute*), 94
 - height
 - insertPage (*Document method*), 120
 - layout (*Document method*), 114
 - newPage (*Document method*), 120
 - open, 101
 - height (*IRect attribute*), 138
 - height (*Pixmap attribute*), 186
 - height (*Quad attribute*), 193
 - height (*Rect attribute*), 197
 - height (*Shape attribute*), 208
 - html
 - Annot.get_text, 87
 - getText, 167
- ## I
- image
 - non-PDF, extract, 17
 - PDF, extract, 17
 - resolution, 16
 - SVG, vector, 22
 - image_profile () (*Tools method*), 225
 - ImageProperties (), 245
 - includePoint () (*Rect method*), 195
 - includeRect () (*Rect method*), 195
 - info (*Annot attribute*), 93
 - inheritable (*built-in variable*), 257
 - insertFont
 - encoding, 164
 - fontbuffer, 164
 - fontfile, 164
 - fontname, 164
 - set_simple, 164
 - insertFont () (*Page method*), 164
 - insertImage
 - examples, 20, 22
 - filename, 165
 - keep_proportion, 165
 - mask, 165
 - oc, 165

- overlay, 165
- pixmap, 165
- rotate, 165
- stream, 165
- insertImage() (*Page method*), 165
- insertLink() (*Page method*), 161
- insertPage (*Document method*)
 - color, 120
 - fontfile, 120
 - fontname, 120
 - fontsize, 120
 - height, 120
 - width, 120
- insertPage() (*Document method*), 120
- insertPDF (*Document method*)
 - annots, 119
 - from_page, 119
 - links, 119
 - rotate, 119
 - show_progress, 119
 - start_at, 119
 - to_page, 119
- insertPDF() (*Document method*), 119
- insertText
 - border_width, 162, 203
 - color, 162, 203
 - encoding, 162, 203
 - fill, 162, 203
 - fill_opacity, 162, 203
 - fontfile, 162, 203
 - fontname, 162, 203
 - fontsize, 162, 203
 - morph, 162, 203
 - oc, 162, 203
 - overlay, 162
 - render_mode, 162, 203
 - rotate, 162, 203
 - stroke_opacity, 162, 203
- insertText() (*Page method*), 162
- insertText() (*Shape method*), 203
- insertTextbox
 - align, 162, 204
 - border_width, 162, 204
 - color, 162, 204
 - encoding, 162, 204
 - expandtabs, 162, 204
 - fill, 162, 204
 - fill_opacity, 162
 - fontfile, 162, 204
 - fontname, 162, 204
 - fontsize, 162, 204
 - morph, 162, 204
 - oc, 162, 204
 - overlay, 162
 - render_mode, 162, 204
 - rotate, 162, 204
 - stroke_opacity, 162
- insertTextbox() (*Page method*), 162
- insertTextbox() (*Shape method*), 204
- interpolate (*Pixmap attribute*), 187
- intersect() (*IRect method*), 137
- intersect() (*Rect method*), 195
- intersects() (*IRect method*), 137
- intersects() (*Rect method*), 196
- invert() (*Matrix method*), 145
- invertIRect
 - examples, 25
- invertIRect() (*Pixmap method*), 184
- IRect (*built-in class*), 136
- irect (*Pixmap attribute*), 186
- irect (*Rect attribute*), 196
- irect_like (*built-in variable*), 257
- is_open (*Annot attribute*), 94
- is_open (*Outline attribute*), 151
- is_signed (*Widget attribute*), 230
- is_wrapped (*Page attribute*), 248
- isClosed (*Document attribute*), 126
- isConvex (*Quad attribute*), 192
- isEmpty (*IRect attribute*), 139
- isEmpty (*Quad attribute*), 193
- isEmpty (*Rect attribute*), 197
- isEncrypted (*Document attribute*), 127
- isExternal (*Link attribute*), 140
- isExternal (*Outline attribute*), 151
- isFormPDF (*Document attribute*), 126
- isInfinite (*IRect attribute*), 138
- isInfinite (*Rect attribute*), 197
- isMap (*linkDest attribute*), 142
- isPDF (*Document attribute*), 126
- isRectangular (*Quad attribute*), 193
- isRectilinear (*Matrix attribute*), 145
- isReflowable (*Document attribute*), 126
- isRepaired (*Document attribute*), 126
- isStream() (*Document method*), 251
- isUri (*linkDest attribute*), 142
- isWritable (*Font attribute*), 135

J

JPEG

- examples, 22

json

- Annot.get_text, 87

- getText, 167

K

keep_proportion

- insertImage, 165

- showPDFpage, 172

kind (*linkDest attribute*), 142

L

lastLocation (*Document attribute*), 128
 lastPoint (*Shape attribute*), 208
 lastPoint (*TextWriter attribute*), 223
 layer_configs () (*Document method*), 103
 layer_ui_configs () (*Document method*), 106
 layout (*Document method*)
 fontsize, 114
 height, 114
 rect, 114
 width, 114
 layout () (*Document method*), 114
 line_ends (*Annot attribute*), 94
 lineCap
 drawBezier, 163
 drawCircle, 163
 drawCurve, 163
 drawLine, 163
 drawOval, 163
 drawPolyline, 163
 drawRect, 164
 drawSector, 163
 drawSquiggle, 163
 drawZigzag, 163
 finish, 206
 lineJoin
 drawBezier, 163
 drawCircle, 163
 drawCurve, 163
 drawLine, 163
 drawOval, 163
 drawPolyline, 163
 drawRect, 164
 drawSector, 163
 drawSquiggle, 163
 drawZigzag, 163
 finish, 206
 Link (*built-in class*), 139
 LINK_FLAG_B_VALID (*built-in variable*), 264
 LINK_FLAG_FIT_H (*built-in variable*), 264
 LINK_FLAG_FIT_V (*built-in variable*), 265
 LINK_FLAG_L_VALID (*built-in variable*), 264
 LINK_FLAG_R_IS_ZOOM (*built-in variable*), 265
 LINK_FLAG_R_VALID (*built-in variable*), 264
 LINK_FLAG_T_VALID (*built-in variable*), 264
 LINK_GOTO (*built-in variable*), 264
 LINK_GOTOR (*built-in variable*), 264
 LINK_LAUNCH (*built-in variable*), 264
 LINK_NAMED (*built-in variable*), 264
 LINK_NONE (*built-in variable*), 264
 LINK_URI (*built-in variable*), 264
 linkDest (*built-in class*), 141

links
 insertPDF (*Document method*), 119
 links () (*Page method*), 161
 ll (*Quad attribute*), 192
 load_annot () (*Page method*), 171
 loadAnnot () (*Page method*), 171
 loadLinks () (*Page method*), 172
 loadPage () (*Document method*), 109
 lr (*Quad attribute*), 192
 lt (*linkDest attribute*), 142

M

make_table (), 242
 makeBookmark () (*Document method*), 108
 mask
 insertImage, 165
 matrix
 get_pixmap, 86
 getPixmap, 98, 170
 getSVGimage, 170
 Matrix (*built-in class*), 143
 matrix_like (*built-in variable*), 257
 MediaBox (*built-in variable*), 257
 MediaBox (*Page attribute*), 176
 MediaBoxSize (*Page attribute*), 176
 metadata (*Document attribute*), 127
 metadataXML () (*Document method*), 125
 morph
 drawBezier, 163
 drawCircle, 163
 drawCurve, 163
 drawLine, 163
 drawOval, 163
 drawPolyline, 163
 drawRect, 164
 drawSector, 163
 drawSquiggle, 163
 drawZigzag, 163
 finish, 206
 insertText, 162, 203
 insertTextbox, 162, 204
 morph () (*IRect method*), 137
 morph () (*Quad method*), 191
 morph () (*Rect method*), 196
 movePage () (*Document method*), 122
 mupdf_display_errors () (*Tools method*), 226
 mupdf_warnings () (*Tools method*), 226

N

n (*Colorspace attribute*), 97
 n (*Pixmap attribute*), 187
 name (*Colorspace attribute*), 97
 name (*Document attribute*), 127
 name (*Font attribute*), 135

- named (*linkDest attribute*), 142
- need_appearances () (*Document method*), 122
- needsPass (*Document attribute*), 126
- newPage (*Document method*)
 - height, 120
 - width, 120
- newPage () (*Document method*), 120
- newShape () (*Page method*), 174
- newWindow (*linkDest attribute*), 142
- next (*Annot attribute*), 93
- next (*Link attribute*), 141
- next (*Outline attribute*), 151
- next (*Widget attribute*), 229
- nextLocation () (*Document method*), 109
- non-PDF
 - extract image, 17
- norm () (*IRect method*), 137
- norm () (*Matrix method*), 144
- norm () (*Point method*), 189
- norm () (*Rect method*), 196
- normalize () (*IRect method*), 137
- normalize () (*Rect method*), 196
- number (*Page attribute*), 177

O

- object (*built-in variable*), 259
- oc
 - drawBezier, 163
 - drawCircle, 163
 - drawCurve, 163
 - drawLine, 163
 - drawOval, 163
 - drawPolyline, 163
 - drawRect, 164
 - drawSector, 163
 - drawSquiggle, 163
 - drawZigzag, 163
 - finish, 206
 - insertImage, 165
 - insertText, 162, 203
 - insertTextbox, 162, 204
- OCCD (*built-in variable*), 259
- OCG (*built-in variable*), 260
- OCMD (*built-in variable*), 260
- OCPD (*built-in variable*), 259
- opacity (*Annot attribute*), 93
- opacity (*TextWriter attribute*), 223
- open
 - Document, 101
 - filename, 101
 - filetype, 101
 - fontsize, 101
 - height, 101
 - rect, 101

- stream, 101
- width, 101
- Outline (*built-in class*), 151
- outline (*Document attribute*), 126
- outlineXref () (*Document method*), 116
- overlay
 - commit, 207
 - drawBezier, 163
 - drawCircle, 163
 - drawCurve, 163
 - drawLine, 163
 - drawOval, 163
 - drawPolyline, 163
 - drawRect, 164
 - drawSector, 163
 - drawSquiggle, 163
 - drawZigzag, 163
 - insertImage, 165
 - insertText, 162
 - insertTextbox, 162
 - showPDFpage, 172

P

- Page (*built-in class*), 154
- page (*built-in variable*), 258
- page (*linkDest attribute*), 142
- page (*Outline attribute*), 151
- page (*Shape attribute*), 208
- page_xref () (*Document method*), 247
- pageCount (*Document attribute*), 127
- pageCropBox () (*Document method*), 110
- pages
 - delete, 58
 - rearrange, 58
- pages () (*Document method*), 110
- pagetree (*built-in variable*), 258
- pageXref () (*Document method*), 110
- PaperRect (), 241
- PaperSize (), 240
- paperSizes, 243
- parent (*Annot attribute*), 93
- parent (*Page attribute*), 177
- Partial Pixmaps, 16
- PDF
 - extract image, 17
 - picture embed, 20
- pdf_catalog () (*Document method*), 248
- pdf_trailer () (*Document method*), 246
- PDFCatalog () (*Document method*), 124
- PDFTrailer () (*Document method*), 125
- permissions (*Document attribute*), 127
- PhotoImage
 - examples, 22
- Photoshop

- examples, 22
- picture
 - embed PDF, 20
- pillowData() (*Pixmap method*), 185
- pillowWrite() (*Pixmap method*), 185
- pixel() (*Pixmap method*), 183
- pixmap
 - insertImage, 165
- Pixmap (*built-in class*), 180
- planishLine(), 243
- Point (*built-in class*), 188
- point_like (*built-in variable*), 257
- popup_rect (*Annot attribute*), 95
- popup_xref (*Annot attribute*), 94
- Postscript
 - examples, 22
- preRotate() (*Matrix method*), 144
- preScale() (*Matrix method*), 144
- preShear() (*Matrix method*), 144
- preTranslate() (*Matrix method*), 144
- previousLocation() (*Document method*), 109

Q

- Quad (*built-in class*), 191
- quad (*IRect attribute*), 138
- quad (*Rect attribute*), 197
- quad_like (*built-in variable*), 257
- quads
 - searchFor, 174

R

- rawdict
 - Annot.get_text, 87
 - getText, 167
- rb (*linkDest attribute*), 142
- readContents() (*Page method*), 249
- reading order
 - text, 29
- rearrange
 - pages, 58
- rect
 - addFreetextAnnot, 155
 - Document, 101
 - layout (*Document method*), 114
 - open, 101
- rect (*Annot attribute*), 93
- Rect (*built-in class*), 194
- rect (*DisplayList attribute*), 99
- rect (*Link attribute*), 140
- rect (*Page attribute*), 177
- rect (*Quad attribute*), 192
- rect (*Shape attribute*), 208
- rect (*TextPage attribute*), 216
- rect (*TextWriter attribute*), 223

- rect (*Widget attribute*), 230
- rect_like (*built-in variable*), 257
- rectangle
 - extract text, 29
- reload_page() (*Document method*), 110
- render_mode
 - insertText, 162, 203
 - insertTextbox, 162, 204
- reset() (*Widget method*), 229
- reset_mupdf_warnings() (*Tools method*), 226
- resolution
 - image, 16
 - zoom, 16
- resolution (*built-in variable*), 259
- resources (*built-in variable*), 258
- rotate
 - addFreetextAnnot, 155
 - convertToPDF (*Document method*), 111
 - insertImage, 165
 - insertPDF (*Document method*), 119
 - insertText, 162, 203
 - insertTextbox, 162, 204
 - setRotation, 172
 - showPDFpage, 172
 - update, 91
- rotation (*Annot attribute*), 93
- rotation (*Page attribute*), 176
- rotationMatrix (*Page attribute*), 176
- round() (*Rect method*), 194
- run() (*DisplayList method*), 98
- run() (*Page method*), 248

S

- samples (*Pixmap attribute*), 186
- save() (*Document method*), 118
- saveIncr() (*Document method*), 119
- script (*Widget attribute*), 230
- script_calc (*Widget attribute*), 231
- script_change (*Widget attribute*), 231
- script_format (*Widget attribute*), 231
- script_stroke (*Widget attribute*), 230
- scrub() (*Document method*), 117
- search() (*TextPage method*), 215
- searchFor
 - flags, 174
 - quads, 174
- searchFor() (*Page method*), 174
- searchPageFor() (*Document method*), 119
- select() (*Document method*), 115
- set_aa_level() (*Tools method*), 226
- set_blendmode() (*Annot method*), 89
- set_border() (*Annot method*), 90
- set_colors() (*Annot method*), 91
- set_flags() (*Annot method*), 90

- set_info() (*Annot method*), 87
 - set_layer_ui_config() (*Document method*), 107
 - set_line_ends() (*Annot method*), 88
 - set_name() (*Annot method*), 89
 - set_oc() (*Annot method*), 88
 - set_oc() (*Document method*), 103
 - set_oc_states() (*Document method*), 106
 - set_ocmd() (*Document method*), 104
 - set_opacity() (*Annot method*), 88
 - set_open() (*Annot method*), 88
 - set_popup() (*Annot method*), 88
 - set_rect() (*Annot method*), 90
 - set_rotation() (*Annot method*), 90
 - set_simple
 - insertFont, 164
 - set_small_glyph_heights() (*Tools method*), 225
 - set_toc_item() (*Document method*), 116
 - set_xml_metadata() (*Document method*), 246
 - setAlpha() (*Pixmap method*), 184
 - setBorder() (*Link method*), 139
 - setColors() (*Link method*), 140
 - setCropBox() (*Page method*), 175
 - setMediaBox() (*Page method*), 175
 - setMetadata() (*Document method*), 115
 - setOrigin() (*Pixmap method*), 184
 - setPixel() (*Pixmap method*), 183
 - setRect
 - examples, 25
 - setRect() (*Pixmap method*), 183
 - setResolution() (*Pixmap method*), 184
 - setRotation
 - rotate, 172
 - setRotation() (*Page method*), 172
 - setTOC() (*Document method*), 115
 - setToC() (*Document method*), 115
 - setXmlMetadata() (*Document method*), 115
 - Shape (*built-in class*), 198
 - show_aa_level() (*Tools method*), 226
 - show_progress
 - insertPDF (*Document method*), 119
 - showPDFpage
 - clip, 172
 - examples, 20, 22
 - keep_proportion, 172
 - overlay, 172
 - rotate, 172
 - showPDFpage() (*Page method*), 172
 - shrink() (*Pixmap method*), 182
 - size (*Pixmap attribute*), 186
 - sRGB_to_pdf(), 241
 - sRGB_to_rgb(), 242
 - start_at
 - insertPDF (*Document method*), 119
 - store_maxsize (*Tools attribute*), 228
 - store_shrink() (*Tools method*), 226
 - store_size (*Tools attribute*), 228
 - stream
 - Document, 101
 - insertImage, 165
 - open, 101
 - stream (*built-in variable*), 259
 - stride (*Pixmap attribute*), 186
 - stroke_opacity
 - drawBezier, 163
 - drawCircle, 163
 - drawCurve, 163
 - drawLine, 163
 - drawOval, 163
 - drawPolyline, 163
 - drawRect, 164
 - drawSector, 163
 - drawSquiggle, 163
 - drawZigzag, 163
 - finish, 206
 - insertText, 162, 203
 - insertTextbox, 162
 - SVG
 - vector image, 22
- ## T
- table
 - extract, 31
 - text
 - Annot.get_text, 87
 - getText, 167
 - reading order, 29
 - rectangle, extract, 29
 - TEXT_ALIGN_CENTER (*built-in variable*), 263
 - TEXT_ALIGN_JUSTIFY (*built-in variable*), 263
 - TEXT_ALIGN_LEFT (*built-in variable*), 263
 - TEXT_ALIGN_RIGHT (*built-in variable*), 263
 - text_color
 - update, 91
 - text_color (*Widget attribute*), 230
 - text_cont (*Shape attribute*), 208
 - TEXT_DEHYPHENATE (*built-in variable*), 263
 - text_font (*Widget attribute*), 230
 - text_fontsize (*Widget attribute*), 230
 - TEXT_INHIBIT_SPACES (*built-in variable*), 263
 - text_length() (*Font method*), 134
 - text_maxlen (*Widget attribute*), 230
 - TEXT_PRESERVE_IMAGES (*built-in variable*), 263
 - TEXT_PRESERVE_LIGATURES (*built-in variable*), 263
 - TEXT_PRESERVE_SPANS (*built-in variable*), 263
 - TEXT_PRESERVE_WHITESPACE (*built-in variable*), 263

text_type (*Widget attribute*), 230
 TextPage (*built-in class*), 214
 textRect (*TextWriter attribute*), 223
 TextWriter (*built-in class*), 221
 tintWith() (*Pixmap method*), 182
 title (*Outline attribute*), 151
 tl (*IRect attribute*), 138
 tl (*Rect attribute*), 196
 to_page
 convertToPDF (*Document method*), 111
 insertPDF (*Document method*), 119
 Tools (*built-in class*), 224
 top_left (*IRect attribute*), 138
 top_left (*Rect attribute*), 196
 top_right (*IRect attribute*), 138
 top_right (*Rect attribute*), 196
 totalcont (*Shape attribute*), 208
 tr (*IRect attribute*), 138
 tr (*Rect attribute*), 196
 transform() (*Point method*), 189
 transform() (*Quad method*), 191
 transform() (*Rect method*), 195
 transformationMatrix (*Page attribute*), 176
 type (*Annot attribute*), 93

U

ufilename
 embeddedFileAdd (*Document method*), 122
 embeddedFileUpd (*Document method*), 124
 update_file, 92
 ul (*Quad attribute*), 192
 unicode_to_glyph_name(), 241
 unicode_to_glyph_name() (*Font method*), 134
 unit (*Point attribute*), 189
 unitvector (*built-in variable*), 259
 update
 blend_mode, 91
 border_color, 91
 cross_out, 91
 fill_color, 91
 fontsize, 91
 rotate, 91
 text_color, 91
 update() (*Annot method*), 91
 update() (*Widget method*), 229
 update_file
 buffer, 92
 desc, 92
 filename, 92
 ufilename, 92
 update_file() (*Annot method*), 92
 updateLink() (*Page method*), 161
 updateObject() (*Document method*), 125
 updateStream() (*Document method*), 125

ur (*Quad attribute*), 192
 uri (*Link attribute*), 140
 uri (*linkDest attribute*), 142
 uri (*Outline attribute*), 151
V
 valid_codepoints() (*Font method*), 133
 vector
 image SVG, 22
 version (*built-in variable*), 262
 VersionBind (*built-in variable*), 262
 VersionDate (*built-in variable*), 262
 VersionFitz (*built-in variable*), 262
 vertices (*Annot attribute*), 94

W

w (*Pixmap attribute*), 186
 Widget (*built-in class*), 229
 widgets() (*Page method*), 162
 width
 drawBezier, 163
 drawCircle, 163
 drawCurve, 163
 drawLine, 163
 drawOval, 163
 drawPolyline, 163
 drawRect, 164
 drawSector, 163
 drawSquiggle, 163
 drawZigzag, 163
 finish, 206
 insertPage (*Document method*), 120
 layout (*Document method*), 114
 newPage (*Document method*), 120
 open, 101
 width (*IRect attribute*), 138
 width (*Pixmap attribute*), 186
 width (*Quad attribute*), 193
 width (*Rect attribute*), 197
 width (*Shape attribute*), 208
 words
 Annot.get_text, 87
 getText, 167
 wrap_contents() (*Page method*), 248
 write() (*Document method*), 119
 writeImage
 examples, 22, 25
 writeImage() (*Pixmap method*), 184
 writePNG() (*Pixmap method*), 185
 writeText() (*Page method*), 162
 writeText() (*TextWriter method*), 223
 wrong
 file extension, 57

X

- x (*Pixmap attribute*), 186
- x (*Point attribute*), 190
- x0 (*IRect attribute*), 138
- x0 (*Rect attribute*), 197
- x1 (*IRect attribute*), 138
- x1 (*Rect attribute*), 197
- xhtml
 - Annot.get_text, 87
 - getText, 167
- xml
 - Annot.get_text, 87
 - getText, 167
- xml_metadata_xref() (*Document method*), 247
- xref (*Annot attribute*), 94
- xref (*built-in variable*), 259
- xref (*Link attribute*), 141
- xref (*Page attribute*), 177
- xref (*Widget attribute*), 230
- xref_length() (*Document method*), 251
- xref_object() (*Document method*), 250
- xrefObject() (*Document method*), 124
- xrefStream() (*Document method*), 125
- xrefStreamRaw() (*Document method*), 125
- xres (*Pixmap attribute*), 187

Y

- y (*Pixmap attribute*), 187
- y (*Point attribute*), 190
- y0 (*IRect attribute*), 138
- y0 (*Rect attribute*), 197
- y1 (*IRect attribute*), 138
- y1 (*Rect attribute*), 197
- yres (*Pixmap attribute*), 187

Z

- zoom, 16
 - resolution, 16