



# PyMuPDF Documentation

*Release 1.14.3*

**Jorj X. McKie**

**Dec 02, 2018**



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Note on the Name <code>fitz</code> . . . . .	2
1.2	License . . . . .	2
1.3	Covered Version . . . . .	2
<b>2</b>	<b>Installation</b>	<b>3</b>
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 . . . . .	6
2.2	Option 2: Install from Binaries . . . . .	6
2.2.1	Step 1: Install from PyPI . . . . .	6
2.2.2	Step 2: Install from GitHub . . . . .	6
2.2.3	MD5 Checksums . . . . .	6
2.2.4	Targeting Parallel Python Installations . . . . .	7
<b>3</b>	<b>Tutorial</b>	<b>9</b>
3.1	Importing the Bindings . . . . .	9
3.2	Opening a Document . . . . .	9
3.3	Some Document Methods and Attributes . . . . .	10
3.4	Accessing Meta Data . . . . .	10
3.5	Working with Outlines . . . . .	10
3.6	Working with Pages . . . . .	11
3.6.1	Inspecting the Links of a Page . . . . .	11
3.6.2	Rendering a Page . . . . .	11
3.6.3	Saving the Page Image in a File . . . . .	12
3.6.4	Displaying the Image in Dialog Managers . . . . .	12
3.6.5	Extracting Text and Images . . . . .	13
3.6.6	Searching for Text . . . . .	13
3.7	PDF Maintenance . . . . .	14
3.7.1	Modifying, Creating, Re-arranging and Deleting Pages . . . . .	14
3.7.2	Joining and Splitting PDF Documents . . . . .	15
3.7.3	Embedding Data . . . . .	15
3.7.4	Saving . . . . .	15
3.8	Closing . . . . .	16
3.9	Further Reading . . . . .	16
<b>4</b>	<b>Classes</b>	<b>17</b>
4.1	Document . . . . .	17
4.1.1	Remarks on <code>select()</code> . . . . .	32

4.1.2	<code>select()</code> Examples	32
4.1.3	<code>setMetadata()</code> Example	33
4.1.4	<code>setToC()</code> Demonstration	33
4.1.5	<code>insertPDF()</code> Examples	34
4.1.6	Other Examples	34
4.2	Outline	35
4.3	Page	36
4.3.1	Adding Page Content	36
4.3.2	Description of <code>getLinks()</code> Entries	50
4.3.3	Notes on Supporting Links	51
4.3.4	Homologous Methods of Document and Page	51
4.4	Pixmap	52
4.4.1	Supported Input Image Types	58
4.4.2	Details on Saving Images with <code>writeImage()</code>	59
4.4.3	Pixmap Code Snippets	59
4.5	Colorspace	62
4.6	Link	62
4.7	<code>linkDest</code>	64
4.8	Matrix	66
4.8.1	Remarks 1	69
4.8.2	Remarks 2	69
4.8.3	Matrix Algebra	69
4.8.4	Examples	70
4.8.5	Shifting	70
4.8.6	Flipping	71
4.8.7	Shearing	72
4.8.8	Rotating	73
4.9	Identity	74
4.10	<code>IRect</code>	75
4.10.1	Remark	77
4.10.2	<code>IRect</code> Algebra	77
4.10.3	Examples	78
4.11	<code>Rect</code>	78
4.11.1	Remark	82
4.11.2	<code>Rect</code> Algebra	82
4.11.3	Examples	83
4.12	<code>Point</code>	84
4.12.1	Remark	86
4.12.2	<code>Point</code> Algebra	86
4.12.3	Examples	86
4.13	<code>Quad</code>	87
4.13.1	Remark	89
4.14	<code>Shape</code>	89
4.14.1	Usage	100
4.14.2	Examples	100
4.14.3	Common Parameters	101
4.15	<code>Annot</code>	103
4.15.1	Example	109
4.16	<code>Widget</code>	110
4.16.1	Standard Fonts for Widgets	112
4.17	Tools	112
4.17.1	Example Session	115

## 5 Operator Algebra for Geometry Objects

117

5.1	General Remarks	117
5.2	Unary Operations	117
5.3	Binary Operations	118
<b>6</b>	<b>Low Level Functions and Classes</b>	<b>119</b>
6.1	Functions	119
6.2	Device	130
6.3	DisplayList	130
6.4	TextPage	132
6.4.1	Dictionary Structure of <code>extractDICT()</code> and <code>extractRAWDICT()</code>	134
6.5	Working together: DisplayList and TextPage	136
6.5.1	Create a DisplayList	136
6.5.2	Generate Pixmap	136
6.5.3	Perform Text Search	137
6.5.4	Extract Text	137
6.5.5	Further Performance improvements	137
<b>7</b>	<b>Constants and Enumerations</b>	<b>139</b>
7.1	Constants	139
7.2	Font File Extensions	140
7.3	Text Alignment	140
7.4	Preserve Text Flags	140
7.5	Link Destination Kinds	141
7.6	Link Destination Flags	141
7.7	Annotation Types	142
7.8	Annotation Flags	144
7.9	Stamp Annotation Icons	145
7.10	Annotation Line End Styles	145
7.11	PDF Form Field Flags	146
7.11.1	Common to all field types	146
7.11.2	Text fields	146
7.11.3	Button fields	147
7.11.4	Choice fields	147
<b>8</b>	<b>Color Database</b>	<b>149</b>
8.1	Function <code>getColor()</code>	149
8.2	Printing the Color Database	150
<b>9</b>	<b>Appendix 1: Performance</b>	<b>151</b>
9.1	Part 1: Parsing	151
9.2	Part 2: Text Extraction	155
9.3	Part 3: Image Rendering	156
<b>10</b>	<b>Appendix 2: Details on Text Extraction</b>	<b>159</b>
10.1	General structure of a TextPage	159
10.2	Plain Text	159
10.3	HTML	160
10.4	Controlling Quality of HTML Output	160
10.5	DICT (or JSON)	161
10.6	RAWDICT	163
10.7	XML	163
10.8	XHTML	164
10.9	Further Remarks	164
10.10	Performance	165

<b>11 Appendix 3: Considerations on Embedded Files</b>	<b>167</b>
11.1 General	167
11.2 MuPDF Support	167
11.3 PyMuPDF Support	167
<b>12 Appendix 4: Assorted Technical Information</b>	<b>169</b>
12.1 PDF Base 14 Fonts	169
12.2 Adobe PDF Reference 1.7	169
12.3 Using Python Sequences as Arguments in PyMuPDF	169
12.4 Ensuring Consistency of Important Objects in PyMuPDF	170
12.5 Design of Method Page.showPDFpage()	172
12.5.1 Purpose and Capabilities	172
12.5.2 Technical Implementation	173
12.6 Redirecting Error and Warning Messages	174
<b>13 Collection of Recipes</b>	<b>175</b>
13.1 Images	175
13.1.1 How to Make Images from Document Pages	175
13.1.2 How to Increase Image Resolution	175
13.1.3 How to Create Partial Pixmaps (Clips)	176
13.1.4 How to Suppress Annotation Images	176
13.1.5 How to Extract Images: Non-PDF Documents	177
13.1.6 How to Extract Images: PDF Documents	177
13.1.7 How to Handle Stencil Masks	178
13.1.8 How to Make one PDF of all your Pictures	179
13.1.9 How to Create Vector Images	182
13.2 Text	183
13.2.1 How to Extract all Document Text	183
13.2.2 How to Extract Text from within a Rectangle	183
13.2.3 How to Extract Text in Natural Reading Order	184
13.2.4 How to Extract Tables from Documents	185
13.2.5 How to Search for and Mark Text	185
13.2.6 How to Insert Text	186
13.3 Annotations	189
13.3.1 How to Add and Modify Annotations	190
13.3.2 How to Mark Text	193
13.3.3 How to Use FreeText	194
13.3.4 How to Use Ink Annotations	195
13.4 General	197
13.4.1 How to Open with a Wrong File Extension	197
13.4.2 How to Embed or Attach Files	197
13.4.3 How to Delete and Re-Arrange Pages	197
13.4.4 How to Join PDFs	198
13.4.5 How to Add Pages	199
13.4.6 How To Dynamically Clean Up Corrupt PDFs	200
13.4.7 How to Split Single Pages	200
13.4.8 How to Combine Single Pages	202
13.4.9 How to Convert Any Document to PDF	203
13.4.10 How to Access Messages Issued by MuPDF	204
13.5 Low-Level Interfaces	205
13.5.1 How to Iterate through the XREF	205
13.5.2 How to Handle Object Streams	206
13.5.3 How to Handle Page Contents	206
13.5.4 How to Access the PDF Catalog Object	207

13.5.5 How to Access XML Metadata . . . . .	207
<b>14 Change Logs</b>	<b>209</b>
14.1 Changes in Version 1.14.3 . . . . .	209
14.2 Changes in Version 1.14.1 . . . . .	209
14.3 Changes in Version 1.14.0 . . . . .	209
14.4 Changes in Version 1.13.19 . . . . .	210
14.5 Changes in Version 1.13.18 . . . . .	210
14.6 Changes in Version 1.13.17 . . . . .	210
14.7 Changes in Version 1.13.16 . . . . .	211
14.8 Changes in Version 1.13.15 . . . . .	211
14.9 Changes in Version 1.13.14 . . . . .	211
14.10Changes in Version 1.13.13 . . . . .	212
14.11Changes in Version 1.13.12 . . . . .	212
14.12Changes in Version 1.13.11 . . . . .	212
14.13Changes in Version 1.13.7 . . . . .	213
14.14Changes in Version 1.13.6 . . . . .	213
14.15Changes in Version 1.13.5 . . . . .	213
14.16Changes in Version 1.13.4 . . . . .	213
14.17Changes in Version 1.13.3 . . . . .	214
14.18Changes in Version 1.13.2 . . . . .	214
14.19Changes in Version 1.13.1 . . . . .	214
14.20Changes in Version 1.13.0 . . . . .	214
14.21Changes in Version 1.12.4 . . . . .	215
14.22Changes in Version 1.12.3 . . . . .	215
14.23Changes in Version 1.12.2 . . . . .	215
14.24Changes in Version 1.12.1 . . . . .	216
14.25Changes in Version 1.12.0 . . . . .	216
14.26Changes in Version 1.11.2 . . . . .	217
14.27Changes in Version 1.11.1 . . . . .	217
14.28Changes in Version 1.11.0 . . . . .	217
14.29Changes in Version 1.10.0 . . . . .	218
14.29.1MuPDF v1.10 Impact . . . . .	218
14.29.2Other Changes compared to Version 1.9.3 . . . . .	219
14.30Changes in Version 1.9.3 . . . . .	220
14.31Changes in Version 1.9.2 . . . . .	220
14.32Changes in Version 1.9.1 . . . . .	221





## INTRODUCTION



**PyMuPDF** is a Python binding for [MuPDF](http://www.mupdf.com/)<sup>1</sup> – “a lightweight PDF and XPS viewer”.

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

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

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

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

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

While PyMuPDF has been available since several years for an earlier version of MuPDF (v1.2, called **fitz-python** then), it was until only mid May 2015, that its creator and a few co-workers decided to elevate it to support current releases of MuPDF (first v1.7a, up to v1.13.0 as of this writing).

PyMuPDF runs and has been tested on Mac, Linux, Windows XP SP2 and up, Python 2.7 through Python 3.7 (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](https://github.com/rk700/PyMuPDF)<sup>3</sup>. We also are registered on [PyPI](https://pypi.org/project/PyMuPDF/)<sup>4</sup>.

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:

---

<sup>1</sup> <http://www.mupdf.com/>

<sup>2</sup> <http://www.sumatrapdfreader.org/>

<sup>3</sup> <https://github.com/rk700/PyMuPDF>

<sup>4</sup> <https://pypi.org/project/PyMuPDF/>

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

For installation details check out the respective chapter.

There exist several [demo](#)<sup>5</sup> and [example](#)<sup>6</sup> 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](#)<sup>7</sup>).

## 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](#)<sup>8</sup>, if you want to create a commercial product with PyMuPDF.

---

## 1.3 Covered Version

This documentation covers PyMuPDF v1.14.3 features as of **2018-12-01 18:33:20**.

---

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

---

---

<sup>5</sup> <https://github.com/rk700/PyMuPDF/tree/master/demo>

<sup>6</sup> <https://github.com/rk700/PyMuPDF/tree/master/examples>

<sup>7</sup> <https://en.wikipedia.org/wiki/MuPDF>

<sup>8</sup> <http://artifex.com/licensing/>

---

## INSTALLATION

Installation generally encompasses downloading and generating PyMuPDF and MuPDF from sources. This process consists of three steps described below under *Option 1: Install from Sources*.

**However**, for popular configurations, binary setups via wheels are available, detailed out under *Option 2: Install from Binaries*. This process is **much faster**, less error-prone and requires the download of only one file (either `.zip` or `.whl`) – no compiler, no Visual Studio, no download of MuPDF, even no download of PyMuPDF.

### 2.1 Option 1: Install from Sources

#### 2.1.1 Step 1: Download PyMuPDF

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

#### 2.1.2 Step 2: Download and Generate MuPDF

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

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

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

#### Applying any Changes or Hot Fixes to MuPDF

On occasion, vital hot fixes or functional enhancements must be applied to MuPDF source before MuPDF should be generated.

Any such files are contained in the `fitz` directory of the PyMuPDF download – their names all start with an underscore `"_"`. Currently (v1.14.0), these files and their copy destination are the following:

- `_mupdf_config.h` – PyMuPDF's configuration to control the binary file size and the inclusion of MuPDF features, see next section. This file must be renamed and replace MuPDF file `/include/mupdf/fitz/config.h`.
- `_error.c` – replaces MuPDF's error module `/source/fitz/error.c`. Our version redirects MuPDF's warnings and errors to devices which PyMuPDF can intercept, so these messages no longer appear on standard output devices of the operating system (STDOUT and STDERR).

- `_pdf_device.c` – replaces MuPDF file `/source/pdf/pdf_device.c`. The original contains a typo which will bring down the Python interpreter when `Document.convertToPDF()` is used.

### Controlling the Binary File Size:

Since version 1.9, MuPDF includes support for many dozens of additional, so-called NOTO (“no TOFU”) fonts for all sorts of alphabets from all over the world like Chinese, Japanese, Korean, Kyrillic, Indonesian, Chinese etc. If you accept MuPDF’s standard here, the resulting binary for PyMuPDF will be very big and easily approach 30 MB. The features actually needed by PyMuPDF in contrast only represent a fraction of this size: about 8-10 MB currently.

To cut off unneeded stuff from your MuPDF version, our suggested version has the following content:

```
#ifndef FZ_CONFIG_H

#define FZ_CONFIG_H

/*
   Enable the following for spot (and hence overprint/overprint
   simulation) capable rendering. This forces FZ_PLOTTERS_N on.
*/
#define FZ_ENABLE_SPOT_RENDERING 1

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

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

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

/*
   Choose whether to enable JavaScript.
   By default JavaScript is enabled both for mutool and PDF interactivity.
```

(continues on next page)

(continued from previous page)

```

*/
/* #define FZ_ENABLE_JS 1 */

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

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

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

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

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

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

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

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

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

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

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

... omitted lines ...
#endif /* FZ_CONFIG_H */

```

**Generate MuPDF now.**

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

Consult additional installation hints on PyMuPDF's [main page](https://github.com/rk700/PyMuPDF/)<sup>9</sup> on Github. Among other things you will find Wiki pages with details on building the Windows binaries or user provided installation experiences.

<sup>9</sup> <https://github.com/rk700/PyMuPDF/>

### 2.1.3 Step 3: Build / Setup PyMuPDF

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

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

Now perform a `python setup.py install`.

## 2.2 Option 2: Install from Binaries

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

Windows binaries provided “on stock” are for Python 32-bit and 64-bit versions.

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

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

### 2.2.1 Step 1: Install from PyPI

If you find the wheel for your platform on PyPI, issue

```
pip install [--upgrade] PyMuPDF
```

and you are done. **Continue with the next chapter of this manual.**

### 2.2.2 Step 2: Install from GitHub

This section applies, if you prefer a ZIP file (Windows only) or if you need a special (bug-fix or pre-release) wheel.

**Download**<sup>10</sup> your Windows, Mac OS or Linux wheel and issue

```
pip install [--upgrade] PyMuPDF-<...>.whl
```

If your platform is Windows you can also download a **zip file**<sup>11</sup>, unzip it to e.g. your Desktop and open a command prompt at the unzipped folder’s directory, which contains `setup.py`. Enter `python setup.py install` (or `py setup.py install` if you have the Python launcher).

### 2.2.3 MD5 Checksums

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

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

---

<sup>10</sup> <https://github.com/rk700/pymupdf/releases>

<sup>11</sup> [https://github.com/JorjMcKie/PyMuPDF-Optional-Material/tree/master/binary\\_setups](https://github.com/JorjMcKie/PyMuPDF-Optional-Material/tree/master/binary_setups)

```
fitz
└─ fitz
   └─ __init__.py
   └─ _fitz.pyd
   └─ fitz.py
   └─ utils.py
└─ MANIFEST
└─ md5.txt
└─ PKG-INFO
└─ setup.py
```

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

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

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

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

## 2.2.4 Targeting Parallel Python Installations

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

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

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

```
py setup.py install
```

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

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





## TUTORIAL

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

Because MuPDF supports not only PDF, but also XPS, OpenXPS, CBZ, CBR, FB2 and EPUB formats, so does PyMuPDF<sup>33</sup>. 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:

```
>>> import fitz
```

You can check your version by printing the docstring:

```
>>> print(fitz.__doc__)
PyMuPDF 1.13.16: Python bindings for the MuPDF 1.13.0 library,
built on 2018-07-26 09:52:26
```

Or simply

```
>>> fitz.version
('1.13.16', '1.13.0', '20180726095226')
```

### 3.2 Opening a Document

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

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

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

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

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

---

<sup>33</sup> PyMuPDF lets you also open several image file types just like normal documents. See section *Supported Input Image Types* in chapter *Pixmap* for more comments.

### 3.3 Some Document Methods and Attributes

Method / Attribute	Description
<code>Document.pageCount</code>	number of pages ( <i>int</i> )
<code>Document.metadata</code>	metadata ( <i>dict</i> )
<code>Document.getToC()</code>	table of contents ( <i>list</i> )
<code>Document.loadPage()</code>	read a page ( <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 Reference 1.7](#) for PDF. Further information can also be found in chapter [Document](#). The meta data fields are strings or `None` if not otherwise indicated. Also be aware that not all of them always contain meaningful data – even if they are not `None`.

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

---

**Note:** Apart from these standard metadata, **PDF documents** starting from PDF version 1.4 may also contain so-called “*metadata streams*”. Information in such streams is coded in XML. PyMuPDF deliberately contains no XML components, so we do not directly support access to information contained therein. But you can extract the stream as a whole, inspect or modify it using a package like `lxml`<sup>12</sup> 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)`<sup>13</sup> resp. `export`<sup>14</sup> metadata from resp. to CSV files.

---

### 3.5 Working with Outlines

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

---

<sup>12</sup> <https://pypi.org/project/lxml/>

<sup>13</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/csv2meta.py>

<sup>14</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/meta2csv.py>

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

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

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

**Note:** There are two utility scripts in the repository that `import (PDF only)`<sup>15</sup> resp. `export`<sup>16</sup> 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.

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

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

`n` may be any positive or negative integer less than `doc.pageCount`. Negative numbers count backwards from the end, so `doc[-1]` is the last page, like with Python sequences.

Some typical uses of *Pages* follow:

### 3.6.1 Inspecting the Links of a Page

Links are shown as “hot areas” when a document is displayed with some 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 and their types.

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

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

### 3.6.2 Rendering a Page

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

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

`pix` is a *Pixmap* object that (in this case) contains an **RGBA** 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,

<sup>15</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/csv2toc.py>

<sup>16</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/toc2csv.py>

rotation, mirroring, shifting, shearing, etc. For example: to create an **RGB** image (i.e. containing no alpha channel), specify `pix = page.getPixmap(alpha = False)`.

---

**Note:** You can also create a **vector** image of a page by using `Page.getSVGImage()`. Refer to this [Wiki](#)<sup>17</sup> for details.

---

### 3.6.3 Saving the Page Image in a File

We can simply store the image in a PNG file:

```
>>> pix.writePNG("page-0.png")
```

### 3.6.4 Displaying the Image in Dialog Managers

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](#)<sup>18</sup> directory.

#### wxPython

Consult their documentation for adjustments to RGB pixmaps and, potentially, specifics for your wxPython release.

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

#### Tkinter

Please also see section 3.19 of the [Pillow documentation](#)<sup>19</sup>, especially for changes when processing RGB pixmaps.

```
>>> from PIL import Image, ImageTk
>>> img = Image.frombytes("RGBA", [pix.width, pix.height], pix.samples)
>>> tking = ImageTk.PhotoImage(img)
```

If your Python contains the **Tk 8.6** version, PNG files and image data are directly supported and you can get along **without using PIL**:

```
>>> pngdata = pix.getPNGData()
>>> tking = tkinter.PhotoImage(data = pngdata)
```

If you are looking for a complete Tkinter script paging through a document, [here it is](#)<sup>20</sup>. You need the [PySimpleGUI](#)<sup>21</sup> pure Python package.

---

<sup>17</sup> <https://github.com/rk700/PyMuPDF/wiki/Vector-Image-Support>

<sup>18</sup> <https://github.com/rk700/PyMuPDF/tree/master/examples>

<sup>19</sup> <https://Pillow.readthedocs.io>

<sup>20</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/doc-browser.py>

<sup>21</sup> <https://pypi.org/project/PySimpleGUI/>

## PyQt4, PyQt5, PySide

Please also see section 3.16 of the [Pillow documentation](#)<sup>22</sup>, especially for changes when processing RGB pixmaps.

```
>>> from PIL import Image, ImageQt
>>> img = Image.frombytes("RGBA", [pix.width, pix.height], pix.samples)
>>> qting = ImageQt.ImageQt(img)
```

### 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("type")
```

Use one of the following strings for "type" to obtain different formats<sup>34</sup>:

- "text": (default) plain text with line breaks. No formatting, no text position details, no images.
- "html": creates a full visual version of the page including any images. This can be displayed with your internet browser.
- "dict": same information level as HTML, but provided as a Python dictionary. See [TextPage.extractDICT\(\)](#) for details of its structure.
- "rawdict": a super-set of [TextPage.extractDICT\(\)](#). It additionally provides character detail information like XML. See [TextPage.extractRAWdict\(\)](#) for details of its structure.
- "xhtml": text information level as the TEXT version but includes images. Can also be displayed by internet browsers.
- "xml": contains no images, but full position and font information down to each single text character. Use an XML module to interpret.

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

### 3.6.6 Searching for Text

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

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

This delivers a list of up to 16 rectangles (see [Rect](#)), each of which surrounds one occurrence of the string "mupdf" (case insensitive). You could use this information to e.g. highlight those areas 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](#)<sup>23</sup> and [demo-lowlevel.py](#)<sup>24</sup>. Among other things they contain details on how the [TextPage](#), [Device](#)

<sup>22</sup> <https://Pillow.readthedocs.io>

<sup>34</sup> [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\(\)](#).

<sup>23</sup> <https://github.com/rk700/PyMuPDF/blob/master/demo/demo.py>

<sup>24</sup> <https://github.com/rk700/PyMuPDF/blob/master/demo/demo-lowlevel.py>

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 files are read-only.

However, you can convert **any document** (including images) to a PDF and then apply all PyMuPDF features to the result of this conversion. Find out more here *Document.convertToPDF()*, and also look at the demo script *pdf-converter.py*<sup>25</sup> which can convert any supported document to PDF.

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

Apart from changes made by you, there are less obvious ways how a PDF may become “modified”:

- During open, integrity checks are used to determine the health of the PDF structure. If errors are encountered, the base library goes a long way to correct them and present a readable document. If this is the case, the document is regarded as being modified.
- After a document has been decrypted, the document in memory has also changed and hence also counts as being modified.

In these two cases, *Document.save()* will store a **repaired**, and (optionally) **decrypted** version<sup>36</sup>, and you must specify a **new file**. Otherwise, you have the option to save your changes as update appendices to the original file (“incremental saves” below), which is very much faster in most cases.

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()* and *Document.movePage()* copy or move a page to other locations within the same document.

These methods are just wrappers for the following more sophisticated method:

*Document.select()* shrinks a PDF down to selected pages. Parameter is a sequence<sup>35</sup> of the page numbers that you want to include. 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,

---

<sup>25</sup> <https://github.com/rk700/PyMuPDF/blob/master/demo/pdf-converter.py>

<sup>36</sup> If the PDF is encrypted, using *doc.save(..., decrypt=False)* will again create an encrypted PDF with the same passwords as the original.

<sup>35</sup> “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.

- 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 opened 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`<sup>26</sup>.

### 3.7.3 Embedding Data

PDFs can be used as containers for arbitrary data (executables, other PDFs, text 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](#)<sup>27</sup>, or the example scripts [embedded-copy.py](#)<sup>28</sup>, [embedded-export.py](#)<sup>29</sup>, [embedded-import.py](#)<sup>30</sup>, and [embedded-list.py](#)<sup>31</sup>.

### 3.7.4 Saving

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

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

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

<sup>26</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/PDFjoiner.py>

<sup>27</sup> <https://github.com/rk700/PyMuPDF/wiki/Dealing-with-Embedded-Files>

<sup>28</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/embedded-copy.py>

<sup>29</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/embedded-export.py>

<sup>30</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/embedded-import.py>

<sup>31</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/embedded-list.py>

Save Option	mutool	Effect
garbage=1	g	garbage collect unused objects
garbage=2	gg	in addition to 1, compact xref tables
garbage=3	ggg	in addition to 2, merge duplicate objects
garbage=4	gggg	in addition to 3, skip duplicate streams
clean=1	c	clean content streams
deflate=1	z	deflate uncompressed streams
ascii=1	a	convert binary data to ASCII format
linear=1	l	create a linearized version
expand=1	i	decompress images
expand=2	f	decompress fonts
expand=255	d	decompress all
incremental=1	n/a	append changes to the original
decrypt=1	n/a	remove passwords

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

## 3.8 Closing

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

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

## 3.9 Further Reading

Also have a look at PyMuPDF’s [Wiki](#)<sup>32</sup> 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.

---

<sup>32</sup> <https://github.com/rk700/PyMuPDF/wiki>



## CLASSES

## 4.1 Document

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

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

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

Method / Attribute	Short Description
<i>Document.authenticate()</i>	decrypt the document
<i>Document.close()</i>	close the document
<i>Document.copyPage()</i>	PDF only: copy a page to another location
<i>Document.convertToPDF()</i>	write a PDF version to memory
<i>Document.deletePage()</i>	PDF only: delete a page by its number
<i>Document.deletePageRange()</i>	PDF only: delete a range of pages
<i>Document.embeddedFileAdd()</i>	PDF only: add a new embedded file from buffer
<i>Document.embeddedFileDel()</i>	PDF only: delete an embedded file entry
<i>Document.embeddedFileGet()</i>	PDF only: extract an embedded file buffer
<i>Document.embeddedFileInfo()</i>	PDF only: metadata of an embedded file
<i>Document.embeddedFileUpd()</i>	PDF only: change an embedded file
<i>Document.embeddedFileSetInfo()</i>	PDF only: change metadata of an embedded file
<i>Document.getPageFontList()</i>	PDF only: make a list of fonts on a page
<i>Document.getPageImageList()</i>	PDF only: make a list of images on a page
<i>Document.getPagePixmap()</i>	create a pixmap of a page by page number
<i>Document.getPageText()</i>	extract the text of a page by page number
<i>Document.getToC()</i>	create a table of contents
<i>Document.insertPage()</i>	PDF only: insert a new page
<i>Document.insertPDF()</i>	PDF only: insert pages from another PDF
<i>Document.layout()</i>	re-paginate the document (if supported)
<i>Document.loadPage()</i>	read a page
<i>Document.movePage()</i>	PDF only: move a page to another location
<i>Document.newPage()</i>	PDF only: insert a new empty page
<i>Document.save()</i>	PDF only: save the document
<i>Document.saveIncr()</i>	PDF only: save the document incrementally
<i>Document.searchPageFor()</i>	search for a string on a page
<i>Document.select()</i>	PDF only: select a subset of pages
<i>Document.setMetadata()</i>	PDF only: set the metadata
<i>Document.setToC()</i>	PDF only: set the table of contents (TOC)
<i>Document.write()</i>	PDF only: writes the document to memory

Continued on next page

Table 1 – continued from previous page

Method / Attribute	Short Description
<i>Document.embeddedFileCount</i>	number of embedded files
<i>Document.FormFonts</i>	PDF only: list of existing field fonts
<i>Document.isClosed</i>	has document been closed?
<i>Document.isPDF</i>	is this a PDF?
<i>Document.isFormPDF</i>	is this a Form PDF?
<i>Document.isReflowable</i>	is this a reflowable document?
<i>Document.metadata</i>	metadata
<i>Document.name</i>	filename of document
<i>Document.needsPass</i>	require password to access data?
<i>Document.isEncrypted</i>	document (still) encrypted?
<i>Document.openErrCode</i>	> 0 if repair occurred during open
<i>Document.openErrMsg</i>	last error message if openErrCode > 0
<i>Document.outline</i>	first <i>Outline</i> item
<i>Document.pageCount</i>	number of pages
<i>Document.permissions</i>	permissions to access the document

## Class API

class Document

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

Creates a Document object.

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

### Parameters

- `filename` (*str/pathlib*) – A UTF-8 string or `pathlib` object containing a file path (or a file type, see below).
- `stream` (*bytes/bytearray*) – A memory area containing a supported document. Its type **must** be specified by either `filename` or `filetype`.
- `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 document types with a variable page layout ("reflowable" documents), like e-books or HTML, and ignored otherwise. If specified, it must be a non-empty, finite rectangle with top-left coordinates (0, 0). Together with parameter `fontsize`, each page will be accordingly laid out and hence also determine the number of pages.
- `width` (*float*) – may used together with `height` as an alternative to `rect` to specify layout information.

- `height` (*float*) – may used together with `width` as an alternative to `rect` to specify layout information.
- `fontsize` (*float*) – the default fontsize for reflowable document types. This parameter is ignored if none of the parameters `rect` or `width` and `height` are specified. Will be used to calculate the page layout.

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

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

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

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

`authenticate(password)`

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

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

**Return type** `int`

**Returns** positive value if decryption was successful, zero otherwise. If successful, indicator `isEncrypted` is set to `False`.

`loadPage(pno = 0)`

Load a [Page](#) for further processing like rendering, text searching, etc.

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

**Return type** [Page](#)

---

**Note:** Conveniently, pages can also be loaded via indexes over the document: `doc.loadPage(n) == doc[n]`. Consequently, a document can also be used as an iterator over its pages, e.g. for `page in doc: ...` and for `page in reversed(doc): ...` will yield the [Pages](#) of `doc` as `page`.

---

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

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

**Parameters**

- `from_page` (*int*) – first page to copy (0-based). Default is first page.

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

#### Return type bytes

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

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

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

---

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

- Image files: perfect, no issues detected. Apparently however, image transparency is ignored. If you need that (like for a watermark), use `Page.insertImage()` instead. Otherwise, this method is recommended for its much better performance.
  - XPS: appearance very good. Links work fine, outlines (bookmarks) are lost, but can easily be recovered<sup>45</sup>.
  - EPUB, CBZ, FB2: similar to XPS.
  - SVG: medium. Roughly comparable to `svglib`<sup>37</sup>.
- 

`getToC(simple = True)`

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

---

<sup>45</sup> 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`<sup>46</sup>.

<sup>46</sup> <https://github.com/rk700/PyMuPDF/blob/master/demo/pdf-converter.py>

<sup>37</sup> <https://github.com/deeplook/svglib>

**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.getPagePixmap\(\)](#).

**Return type** [Pixmap](#)

`getPageImageList(pno)`

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

**Parameters** `pno (int)` – page number, 0-based, any value `< len(doc)`.

**Return type** `list`

#### Returns

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

- `xref (int)` is the image object number,
- `smask (int optional)` is the object number of its soft-mask image (if present),
- `width` and `height (ints)` are the image dimensions,
- `bpc (int)` denotes the number of bits per component (a typical value is 8),
- `colorspace (str)` a string naming the colorspace (like `DeviceRGB`),
- `alt. colorspace (str optional)` is any alternate colorspace depending on the value of `colorspace`,
- `name (str)` is the symbolic name by which the **page references the image** in its content stream, and
- `filter (str optional)` is the decode filter of the image ([Adobe PDF Reference 1.7](#), pp. 65).

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)
[[316, 0, 261, 115, 8, 'DeviceRGB', '', 'Im1', 'DCTDecode']]
>>> pix = fitz.Pixmap(doc, 316)      # 316 is the xref of the image
```

(continues on next page)

(continued from previous page)

```
>>> pix
fitz.Pixmap(DeviceRGB, fitz.IRect(0, 0, 261, 115), 0)
```

---

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

---

`getPageFontList(pno)`

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

**Parameters** `pno` (*int*) – page number, 0-based, any value < `len(doc)`.

**Return type** `list`

**Returns**

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

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

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

---

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

---

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

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

**Parameters**

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

**Return type** `str`

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

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

#### Parameters

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

`select(s)`

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

**Parameters** `s` (*sequence*) – A 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.

`setMetadata(m)`

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

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

`setToC(toc)`

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

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

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

- `[lvl, title, page, dest]`, where
  - `lvl` is the hierarchy level (int > 0) of the item, starting with 1 and being at most 1 higher than that of the predecessor,

- `title` (`str`) is the title to be displayed. It is assumed to be UTF-8-encoded (relevant for multibyte code points only).
- `page` (`int`) is the target page number (**attention: 1-based to support `getToC()`-output**), must be in valid page range if positive. Set this to `-1` if there is no target, or the target is external.
- `dest` (optional) is a dictionary or a number. If a number, it will be interpreted as the desired height (in points) this entry should point to on page in the current document. Use a dictionary (like the one given as output by `getToC(simple = False)`) if you want to store destinations that are either “named”, or reside outside this document (other files, internet resources, etc.).

**Return type** `int`

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

---

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

---

`save(outfile, garbage=0, clean=False, deflate=False, incremental=False, ascii=False, expand=0, linear=False, pretty=False, decrypt=True)`  
PDF only: Saves the document in its **current state** under the name `outfile`.

#### Parameters

- `outfile` (`str`) – The file name to save to. Must be different from the original value if “incremental” is false or zero. When saving incrementally, “garbage” and “linear” **must be** false or zero and this parameter **must equal** the original filename (for convenience use `doc.name`).
- `garbage` (`int`) – Do garbage collection. Positive values exclude incremental.
  - 0 = none
  - 1 = remove unused objects
  - 2 = in addition to 1, compact xref table
  - 3 = in addition to 2, merge duplicate objects
  - 4 = in addition to 3, check object streams for duplication (may be slow)
- `clean` (`bool`) – Clean content streams<sup>44</sup>.
- `deflate` (`bool`) – Deflate (compress) uncompressed streams.
- `incremental` (`bool`) – Only save changed objects. Excludes “garbage” and “linear”. Cannot be used for decrypted files and for repaired files (`openErrCode > 0`). In these cases saving to a new file is required.
- `ascii` (`bool`) – Where possible convert binary data to ASCII.
- `expand` (`int`) – Decompress objects. Generates versions that can be better read by some other programs.
  - 0 = none

---

<sup>44</sup> Content streams describe what (e.g. text or images) appears where and how on a page. PDF uses a specialized mini language similar to PostScript to do this (pp. 985 in *Adobe PDF Reference 1.7*), which gets interpreted when a page is loaded.



- 1 = images
- 2 = fonts
- 255 = all
- `linear (bool)` – Save a linearised version of the document. This option creates a file format for improved performance when read via internet connections. Excludes “incremental”.
- `pretty (bool)` – Prettify the document source for better readability.
- `decrypt (bool)` – Save a decrypted copy (the default). If false, the resulting PDF will be encrypted with the same password as the original. Will be ignored for non-encrypted files.

`saveIncr()`

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

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

`searchPageFor(pno, text, hit_max = 16, quads = False)`

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

`write(garbage=0, clean=False, deflate=False, ascii=False, expand=0, linear=False, pretty=False, decrypt=True)`

PDF only: Writes the **current content of the document** to a bytes object instead of to a file like `save()`. Obviously, you should be wary about memory requirements. The meanings of the parameters exactly equal those in `save()`. Cpater *Collection of Recipes* contains an example for using this method as a pre-processor to `pdfwr`<sup>38</sup>.

**Return type** bytes

**Returns** a bytes object containing the complete document data.

`insertPDF(docsrc, from_page = -1, to_page = -1, start_at = -1, rotate = -1, links = True)`

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

#### Parameters

- `docsrc (Document)` – An opened PDF Document which must not be the current document object. However, it may refer to the same underlying file.
- `from_page (int)` – First page number in `docsrc`. Default is zero.
- `to_page (int)` – Last page number in `docsrc` to copy. Default is the last page.
- `start_at (int)` – First copied page will become page number `start_at` in the destination. If omitted, the page range will be appended to current document. If zero, the page range will be inserted before current first page.

<sup>38</sup> <https://pypi.python.org/pypi/pdfwr/0.3>

- `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 with the copy. Default is `True`. An **internal** link is always excluded if its destination is outside the copied page range.

---

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

---

---

**Note:** `docsrc` bookmarks **will not be copied**. It is easy however, to recover a table of contents for the resulting document. Look at the examples below and at program [PDFjoiner.py](#)<sup>39</sup> in the *examples* directory: it can join PDF documents and at the same time piece together respective parts of the tables of contents.

---

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

PDF only: Insert a new page. Default page dimensions are those of A4 portrait paper format. Optionally, text can also be inserted – provided as a string or as a sequence.

#### Parameters

- `to (int)` – page number (0-based) in front of which to insert. Valid specifications must be in range `-1 <= pno <= len(doc)`. The default `-1` and `pno = len(doc)` indicate end of document, i.e. after the last page.
- `text (str or sequence)` – optional text to put on the page. If given, it will start at 72 points (one inch) below top and 50 points from left. Line breaks (`\n`) will be honored, if it is a string. No care will be taken as to whether lines are too wide. However, text output stops when no more lines will fit on the page (discarding any remaining text). If a sequence is specified, its entries must be a of type string. Each entry will be put on one line. Line breaks *within an entry* will be treated as any other white space. If you want to calculate the number of lines fitting on a page beforehand, use this formula: `int((height - 108) / (fontsize * 1.2))`. So, this methods reserves one inch at the top and 1/2 inches at the bottom of the page as free space.
- `fontsize (float)` – font size in pixels. Default is 11. If more than one line is provided, a line spacing of `fontsize * 1.2` (fontsize plus 20%) is used.
- `width (float)` – width in pixels. Default is 595 (A4 width). Choose 612 for *Letter width*.
- `height (float)` – page height in pixels. Default is 842 (A4 height). Choose 792 for *Letter height*.
- `fontname (str)` – name of one of the [PDF Base 14 Fonts](#) (default is “Helvetica”) if `fontfile` is not specified.
- `fontfile (str)` – file path of a font existing on the system. If this parameter is specified, specifying `fontname` is **mandatory**. If the font is new to the PDF, it will be embedded. Of the font file, index 0 is used. Be sure to choose a font that supports horizontal, left-to-right spacing.

---

<sup>39</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/PDFjoiner.py>

- `color (sequence)` – RGB text color specified as a triple of floats in range 0 to 1. E.g. specify black (default) as (0, 0, 0), red as (1, 0, 0), some gray value as (0.5, 0.5, 0.5), etc.

**Return type** `int`

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

**Notes:**

This method can be used to

1. create a PDF containing only one empty page of a given dimension. The size of such a file is well below 500 bytes and hence close to the theoretical PDF minimum.
2. create a protocol page of which files have been embedded, or separator pages between joined pieces of PDF Documents.
3. convert textfiles to PDF like in the demo script [text2pdf.py](#)<sup>40</sup>.
4. For now, the inserted text should restrict itself to one byte character codes.
5. An easy way to create pages with a usual paper format, use a statement like `width, height = fitz.PaperSize("A4-L")`.
6. To simplify color specification, we provide a [Color Database](#). This allows you to specify `color = getColor("turquoise")`, without bothering about any more details.

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

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

If you do not need to insert text with your new page right away, then this method is the more convenient one: it saves you one statement if you need it for subsequent work – see the below example.

**Return type** `Page`

**Returns** the page object just inserted.

```
>>> # let the following be a list of image files, from which we
>>> # create a PDF with one image per page:
>>> imglist = [...] # list of image filenames
>>> doc = fitz.open() # new empty PDF
>>> for img in imglist:
>>>     pix = fitz.Pixmap(img)
>>>     page = doc.newPage(-1, width = pix.width, height = pix.height)
>>>     page.insertImage(page.rect, pixmap = pix)
>>> doc.save("image-file.pdf", deflate = True)
```

`deletePage(pno = -1)`

PDF only: Delete a page given by its 0-based number in range  $0 \leq pno < \text{len}(\text{doc})$ .

**Parameters** `pno (int)` – the page to be deleted. For -1 the last page will be deleted.

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

PDF only: Delete a range of pages specified as 0-based numbers. Any -1 parameter will first be replaced by  $\text{len}(\text{doc}) - 1$ . After that, condition  $0 \leq \text{from\_page} \leq \text{to\_page} < \text{len}(\text{doc})$  must be true. If the parameters are equal, one page will be deleted.

<sup>40</sup> <https://github.com/rk700/PyMuPDF/blob/master/demo/text2pdf.py>

**Parameters**

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

`copyPage(pno, to = -1)`

PDF only: Copy a page within the document.

**Parameters**

- `pno (int)` – the page to be copied. Must be in range  $0 \leq pno < \text{len}(\text{doc})$ .
- `to (int)` – the page number in front of which to copy. To insert after the last page (default), specify -1.

`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  $0 \leq pno < \text{len}(\text{doc})$ .
- `to (int)` – the page number in front of which to insert the moved page. To insert after the last page (default), specify -1.

`embeddedFileAdd(buffer, name, 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).

**Parameters**

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

---

**Note:** The position of the new entry in the embedded files list can in general not be predicted. Do not assume a specific place (like the end or the beginning), even if the chosen name seems to suggest it. If you add several files, their sequence in that list will probably not be maintained either. In addition, the various PDF viewers each seem to use their own ordering logic when showing the list of embedded files for the same PDF.

---

`embeddedFileGet(n)`

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

**Parameters** `n (int/str)` – index or name of entry. For an integer  $0 \leq n < \text{embeddedFileCount}$  must be true.

**Return type** bytes

`embeddedFileDel(name)`

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

**Parameters** *name (str)* – name of entry. We do not support entry **numbers** for this function yet. If you need to e.g. delete **all** embedded files, scan through embedded files by number, and use the returned dictionary's *name* entry to delete each one.

**Return type** *int*

**Returns** the number of deleted file entries.

**Caution:** This function will delete **every entry with this name**. Be aware that PDFs not created with PyMuPDF may contain duplicate names, in which case more than one entry may be deleted.

`embeddedFileInfo(n)`

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

**Parameters** *n (int/str)* – index or name of entry. For an integer  $0 \leq n < \text{embeddedFileCount}$  must be true.

**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

`embeddedFileUpd(n, 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**

- *n (int/str)* – index or name of entry. For an integer  $0 \leq n < \text{embeddedFileCount}$  must be true.
- *buffer (bytes/bytearray)* – the new file content.
- *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. For an integer  $0 \leq n < \text{embeddedFileCount}$  must be true.
- *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 next version.

---

`close()`

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

`outline`

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

**Type** [Outline](#)

`isClosed`

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

**Type** `bool`

`isPDF`

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

**Type** `bool`

`isFormPDF`

`True` if this is a Form PDF document with field count greater zero, else `False`.

**Type** `bool`

`isReflowable`

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

**Type** `bool`

`needsPass`

Contains an indicator showing whether the document is encrypted (`True`) or not (`False`). 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 an authentication, it is set to `False` to reflect the situation.

**Type** `bool`

`permissions`

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

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

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

**Type** dict

**metadata**

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

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

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

**Type** dict

**name**

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

**Type** str

**pageCount**

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

**Type** int

**openErrCode**

If `openErrCode > 0`, errors have occurred while opening / parsing the document, which usually means damages like document structure issues. In this case **incremental** save cannot be used. The **document is available** for processing however, potentially with restrictions (depending on damage details).

**Type** int

**openErrMsg**

Contains either an empty string or the last open error message if `openErrCode > 0`. To see all messages, look at [Tools.fitz\\_stderr](#), e.g. `print(fitz.TOOLS.fitz_stderr)`.

**Type** str

**embeddedFileCount**

Contains the number of files in the `/EmbeddedFiles` list, -1 if the document is not a PDF.

**Type** int

FormFonts

A list of font resource names. Contains None if not a PDF and [] if not a Form PDF.

**Type** int

---

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

---

#### 4.1.1 Remarks on `select()`

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, re-arrange a document from back to front, duplicate it, and so forth. In combination with text search or extraction you can also omit / include pages with no text or containing a certain text, etc.

If you have de-selected many pages, consider specifying the `garbage` option to eventually reduce the resulting document's size (when saving to a new file).

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

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

#### 4.1.2 `select()` Examples

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

Delete pages with no text:

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

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

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

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

Create a sub document with only the odd pages:



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

Concatenate a document with itself:

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

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

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

### 4.1.3 setMetadata() Example

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

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

### 4.1.4 setToC() Demonstration

This shows how to modify or add a table of contents. Also have a look at [csv2toc.py](https://github.com/rk700/PyMuPDF/blob/master/examples/csv2toc.py)<sup>41</sup> and [toc2csv.py](https://github.com/rk700/PyMuPDF/blob/master/examples/toc2csv.py)<sup>42</sup> in the examples directory.

<sup>41</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/csv2toc.py>

<sup>42</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/toc2csv.py>

```

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

```

### 4.1.5 insertPDF() Examples

#### (1) Concatenate two documents including their TOCs:

```

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

```

Obviously, similar ways can be found in more general situations. Just make sure that hierarchy levels in a row do not increase by more than one. Inserting dummy bookmarks before and after `toc2` segments would heal such cases. A ready-to-use GUI (wxPython) solution can be found in script [PDFjoiner.py](#)<sup>43</sup> of the examples directory.

#### (2) More examples:

```

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

```

```

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

```

```

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

```

### 4.1.6 Other Examples

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

<sup>43</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/PDFjoiner.py>

```

for i in range(len(doc)):
    imglist = doc.getPageImageList(i)
    for img in imglist:
        xref = img[0]                # xref number
        pix = fitz.Pixmap(doc, xref)  # make pixmap from image
        if pix.n - pix.alpha < 4:    # can be saved as PNG
            pix.writePNG("p%s-%s.png" % (i, xref))
        else:                        # CMYK: must convert first
            pix0 = fitz.Pixmap(fitz.csRGB, pix)
            pix0.writePNG("p%s-%s.png" % (i, xref))
            pix0 = None               # free Pixmap resources
        pix = None                   # free Pixmap resources

```

### Rotate all pages of a PDF:

```
>>> for page in doc: page.setRotation(90)
```

## 4.2 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
<i>Outline.down</i>	next item downwards
<i>Outline.next</i>	next item same level
<i>Outline.page</i>	page number (0-based)
<i>Outline.title</i>	title
<i>Outline.uri</i>	string further specifying the outline target
<i>Outline.isExternal</i>	target is outside this document
<i>Outline.is_open</i>	whether sub-outlines are open or collapsed
<i>Outline.isOpen</i>	whether sub-outlines are open or collapsed
<i>Outline.dest</i>	points to link destination details

### Class API

```
class Outline
```

down

The next outline item on the next level down. Is None if the item has no kids.

**Type** *Outline*

next

The next outline item at the same level as this item. Is None if this is the last one in its level.

**Type** *Outline*

page

The page number (0-based) this bookmark points to.

**Type** int

`title`  
The item's title as a string or `None`.  
**Type** `str`

`is_open`  
Or `isOpen` – an indicator showing whether any sub-outlines should be expanded (`True`) or be collapsed (`False`). This information should be interpreted by PDF display software accordingly.  
**Type** `bool`

`isExternal`  
A `bool` specifying whether the target is outside (`True`) of the current document.  
**Type** `bool`

`uri`  
A string specifying the link target. The meaning of this property should be evaluated in conjunction with `isExternal`. The value may be `None`, in which case `isExternal == False`. If `uri` starts with `file://`, `mailto:`, or an internet resource name, `isExternal` is `True`. In all other cases `isExternal == False` and `uri` points to an internal location. In case of PDF documents, this should either be `#nnnn` to indicate a 1-based (!) page number `nnnn`, or a named location. The format varies for other document types, e.g. `uri = '../FixedDoc.fdoc#PG_21_LNK_84'` for page number 21 (1-based) in an XPS document.  
**Type** `str`

`dest`  
The link destination details object.  
**Type** `linkDest`

## 4.3 Page

Class representing a document page. A page object is created by `Document.loadPage()` or, equivalently, via indexing the document like `doc[n]` - it has no independent constructor.

There is a parent-child relationship between a document and its pages. If the document is closed or deleted, all page objects (and their respective children, too) in existence will become unusable. If a page property or method is being used, an exception is raised saying that the page object is “orphaned”.

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

### 4.3.1 Adding Page Content

This is available for PDF documents only. There are basically two groups of methods:

1. Methods making **permanent** changes. This group contains `insertText()`, `insertTextbox()` and all `draw*()` methods. They provide “stand-alone”, shortcut versions for the same-named methods of the `Shape` class. For detailed descriptions have a look in that chapter. Some remarks on the relationship between the `Page` and `Shape` methods:
  - In contrast to `Shape`, the results of page methods are not interconnected: they do not share properties like colors, line width / dashed, morphing, etc.
  - Each page `draw*()` method invokes a `Shape.finish()` and then a `Shape.commit()` and consequently accepts the combined arguments of both these methods.

- Text insertion methods (`insertText()` and `insertTextbox()`) do not need `Shape.finish()` and therefore only invoke `Shape.commit()`.
2. Methods adding **annotations**. Annotations can be added, modified and deleted without necessarily having full document permissions. Their effect is **not permanent** in the sense, that manipulating them does not require to rebuild the document. **Adding** and **deleting** annotations are page methods. **Changing** existing annotations is possible via methods of the `Annot` class.

Method / Attribute	Short Description
<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
<code>Page.addStampAnnot()</code>	PDF only: add a “rubber stamp” annotation
<code>Page.addSquigglyAnnot()</code>	PDF only: add a “squiggly” annotation
<code>Page.addStrikeoutAnnot()</code>	PDF only: add a “strike-out” annotation
<code>Page.addTextAnnot()</code>	PDF only: add comment and a note icon
<code>Page.addUnderlineAnnot()</code>	PDF only: add an “underline” annotation
<code>Page.addWidget()</code>	PDF only: add a PDF Form field
<code>Page.bound()</code>	rectangle of the page
<code>Page.deleteAnnot()</code>	PDF only: delete an annotation
<code>Page.deleteLink()</code>	PDF only: delete a link
<code>Page.drawBezier()</code>	PDF only: draw a cubic Bézier curve
<code>Page.drawCircle()</code>	PDF only: draw a circle
<code>Page.drawCurve()</code>	PDF only: draw a special Bézier 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.getFontList()</code>	PDF only: get list of used fonts
<code>Page.getImageList()</code>	PDF only: get list of used images
<code>Page.getLinks()</code>	get all links
<code>Page.getPixmap()</code>	create a <i>Pixmap</i>
<code>Page.getSVGImage()</code>	create a page image in SVG format
<code>Page.getText()</code>	extract the page’s text
<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 new link
<code>Page.insertText()</code>	PDF only: insert text
<code>Page.insertTextbox()</code>	PDF only: insert a text box
<code>Page.loadLinks()</code>	return the first link on a page
<code>Page.newShape()</code>	PDF only: start a new <i>Shape</i>
<code>Page.searchFor()</code>	search for a string
<code>Page.setCropBox()</code>	PDF only: modify the visible page

Continued on next page

Table 2 – continued from previous page

Method / Attribute	Short Description
<i>Page.setRotation()</i>	PDF only: set page rotation
<i>Page.showPDFpage()</i>	PDF only: display PDF page image
<i>Page.updateLink()</i>	PDF only: modify a link
<i>Page.CropBox</i>	the page's /CropBox
<i>Page.CropBoxPosition</i>	displacement of the /CropBox
<i>Page.firstAnnot</i>	first <i>Annot</i> on the page
<i>Page.firstLink</i>	first <i>Link</i> on the page
<i>Page.MediaBox</i>	the page's /MediaBox
<i>Page.MediaBoxSize</i>	bottom-right point of /MediaBox
<i>Page.number</i>	page number
<i>Page.parent</i>	owning document object
<i>Page.rect</i>	rectangle (mediabox) of the page
<i>Page.rotation</i>	PDF only: page rotation
<i>Page.xref</i>	PDF cross reference number

## Class API

class Page

`bound()`

Determine the rectangle (before transformation) of the page. Same as property *Page.rect* below. For PDF documents this **usually** also coincides with objects /MediaBox and /CropBox, but not always. The best description hence is probably “/CropBox, transformed such that top-left coordinates are (0, 0)”. Also see attributes *Page.CropBox* and *Page.MediaBox*.

**Return type** *Rect*

`addTextAnnot(point, text)`

PDF only: Add a comment icon (“sticky note”) with accompanying text.

**Parameters**

- *point* (*point-like*) – the top left point of a 18 x 18 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.

**Return type** *Annot*

**Returns** the created annotation. Use methods of *Annot* to make any changes.



## 'Text' annotation

`addFreetextAnnot(rect, text, fontsize = 12, fontname = "Helvetica", color = (0, 0, 0), rotate = 0)`

PDF only: Add text in a given rectangle.

**Parameters**

- *rect* (*rect-like*) – the rectangle into which the text should be inserted.
- *text* (*str*) – the text. May contain any Latin characters.
- *fontsize* (*float*) – the font size. Default is 12.

- `fontname (str)` – the font name. Default is “Helvetica”. Accepted alternatives are “Courier”, “Times-Roman”, “ZapfDingbats” and “Symbol”. The name may be abbreviated to the first two characters, “Co” for “Courier”. Lower case is also accepted.
- `color (sequence)` – the text and rectangle border color. Default is black. Text is automatically wrapped to a new line at box width. Lines not fitting to the box will be invisible.
- `rotate (int)` – the text orientation. Accepted values are 0, 90, 270, else zero is used.

**Return type** *Annot*

**Returns** the created annotation. The text and rectangle border will be drawn in the same specified color. Rectangle background is white. These properties can only be changed using special parameters of *Annot.update()*. Changeable properties are text color, box interior and border color and text font size.

`addFileAnnot(pos, buffer, filename, ufilename = None, desc = None)`

PDF only: Add a file attachment annotation with a “PushPin” icon at the specified location.

**Parameters**

- `pos (point-like)` – the top-left point of a 18x18 rectangle containing the MuPDF-provided “PushPin” icon.
- `buffer (bytes/bytearray)` – the data to be stored (actual file content, calculated data, etc.).
- `filename (str)` – the filename.
- `ufilename (str)` – the optional PDF unicode filename. Defaults to filename.
- `desc (str)` – an optional description of the file. Defaults to filename.

**Return type** *Annot*

**Returns** the created annotation. Use methods of *Annot* to make any changes.



**'FileAttachment' annotation**

`addInkAnnot(list)`

PDF only: Add a “freehand” scribble annotation.

**Parameters** `list (sequence)` – a list of one or more lists, each containing point-like items. Each item in these sublists is interpreted as a *Point* through which a connecting line is drawn. Separate sublists thus represent separate drawing lines.

**Return type** *Annot*

**Returns** the created annotation in default appearance (black line of width 1). Use annotation methods with a subsequent *Annot.update()* to modify.

`addLineAnnot(p1, p2)`

PDF only: Add a line annotation.

**Parameters**

- `p1 (point-like)` – the starting point of the line.
- `p2 (point-like)` – the end point of the line.

**Return type** [Annot](#)

**Returns** the created annotation. It is drawn with line color black and line width 1. To change, or attach other information (like author, creation date, line properties, colors, line ends, etc.) use methods of [Annot](#). The **rectangle** is automatically created to contain both points, each one surrounded by a circle of radius 3 ( $= 3 * \text{line width}$ ) to make room for any line end symbols. Use methods of [Annot](#) to make any changes.

```
addRectAnnot(rect)
```

```
addCircleAnnot(rect)
```

PDF only: Add a rectangle, resp. circle annotation.

**Parameters** `rect` (*rect-like*) – the rectangle in which the circle or rectangle is drawn, must be finite and not empty. If the rectangle is not equal-sided, an ellipse is drawn.

**Return type** [Annot](#)

**Returns** the created annotation. It is drawn with line color black, no fill color and line width 1. Use methods of [Annot](#) to make any changes.

```
addPolylineAnnot(points)
```

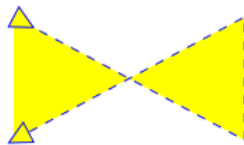
```
addPolygonAnnot(points)
```

PDF only: Add an annotation consisting of lines which connect the given points. A **Polygon's** first and last points are automatically connected, which does not happen for a **PolyLine**. The **rectangle** is automatically created as the smallest rectangle containing the points, each one surrounded by a circle of radius 3 ( $= 3 * \text{line width}$ ). 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:



'PolyLine' annotation

```
addUnderlineAnnot(rect)
```

```
addStrikeoutAnnot(rect)
```

```
addSquigglyAnnot(rect)
```

```
addHighlightAnnot(rect)
```

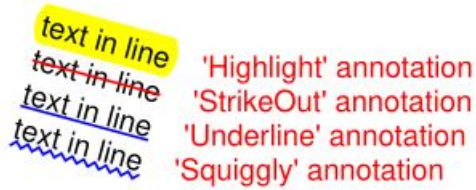
PDF only: These annotations are used for marking some text that has previously been located via `searchFor()`. Colors are automatically chosen: yellowish for highlighting, red for strike out and blue for underlining. Note that `searchFor()` now supports quadrilaterals as an output option. Correspondingly, the `rect` parameter for these annotations may either be rectangles or quadrilaterals.

**Parameters** `rect` (*rect-like/quad-like*) – the rectangle or quad containing the to-be-marked text.

**Return type** [Annot](#)



**Returns** the created annotation. Per annot type, certain color decisions are being made (e.g. “red” for ‘StrikeOut’, “yellow” for ‘Highlight’). To change them, set the “stroke” color accordingly ([Annot.setColors\(\)](#)) and then perform an [Annot.update\(\)](#).



`addStampAnnot(rect, stamp = 0)`

PDF only: Add a “rubber stamp” like annotation to e.g. indicate the document’s intended use (“DRAFT”, “CONFIDENTIAL”, etc.).

#### Parameters

- `rect` (*rect-like*) – rectangle where to place the annotation.
- `stamp` (*int*) – id number of the stamp text. For available stamps see [Stamp Annotation Icons](#).

**Note:** The stamp’s text (e.g. “APPROVED”) and its border line will automatically be sized and put centered in the given rectangle. The property [Annot.rect](#) is automatically calculated to fit and will usually be smaller than this parameter. The appearance can be changed using [Annot.setOpacity\(\)](#) and by setting the “stroke” color (no “fill” color supported).



**'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.

**Note:** Make sure to use parameter `clean = True` when saving the file. This will cause recalculation of the annotations appearance.

`deleteAnnot(annot)`

PDF only: Delete the specified annotation from the page and return the next one.

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

**Return type** [Annot](#)

**Returns** the annotation following the deleted one.

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

`insertText(point, text, fontsize = 11, fontname = "helv", fontfile = None, idx = 0, color = (0, 0, 0), rotate = 0, morph = None, overlay = True)`

PDF only: Insert text starting at point-like point. See `Shape.insertText()`.

`insertTextbox(rect, buffer, fontsize = 11, fontname = "helv", fontfile = None, idx = 0, color = (0, 0, 0), expandtabs = 8, align = TEXT_ALIGN_LEFT, charwidths = None, rotate = 0, morph = None, overlay = True)`

PDF only: Insert text into the specified rect-like rect. See `Shape.insertTextbox()`.

`drawLine(p1, p2, color = (0, 0, 0), width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`

PDF only: Draw a line from p1 to p2 (point-likes). See `Shape.drawLine()`.

`drawZigzag(p1, p2, breadth = 2, color = (0, 0, 0), width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`

PDF only: Draw a zigzag line from p1 to p2 (point-likes). See `Shape.drawZigzag()`.

`drawSquiggle(p1, p2, breadth = 2, color = (0, 0, 0), width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`

PDF only: Draw a squiggly (wavy, undulated) line from p1 to p2 (point-likes). See `Shape.drawSquiggle()`.

`drawCircle(center, radius, color = (0, 0, 0), fill = None, width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`

PDF only: Draw a circle around center (point-like) with a radius of radius. See `Shape.drawCircle()`.

`drawOval(rect, color = (0, 0, 0), fill = None, width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`

PDF only: Draw an oval (ellipse) within the given rectangle (rect-like). See `Shape.drawOval()`.

`drawSector(center, point, angle, color = (0, 0, 0), fill = None, width = 1, dashes = None, roundCap = True, fullSector = True, overlay = True, closePath = False, morph = None)`

PDF only: Draw a circular sector, optionally connecting the arc to the circle's center (like a piece of pie). See `Shape.drawSector()`.

`drawPolyline(points, color = (0, 0, 0), fill = None, width = 1, dashes = None, roundCap = True, overlay = True, closePath = False, morph = None)`

PDF only: Draw several connected lines defined by a sequence of point-likes. See [Shape.drawPolyline\(\)](#).

`drawBezier(p1, p2, p3, p4, color = (0, 0, 0), fill = None, width = 1, dashes = None, roundCap = True, overlay = True, closePath = False, morph = None)`

PDF only: Draw a cubic Bézier curve from p1 to p4 with the control points p2 and p3 (all are point-likes). See [Shape.drawBezier\(\)](#).

`drawCurve(p1, p2, p3, color = (0, 0, 0), fill = None, width = 1, dashes = None, roundCap = True, overlay = True, closePath = False, morph = None)`

PDF only: This is a special case of `drawBezier()`. See [Shape.drawCurve\(\)](#).

`drawRect(rect, color = (0, 0, 0), fill = None, width = 1, dashes = None, roundCap = True, overlay = True, morph = None)`

PDF only: Draw a rectangle. See [Shape.drawRect\(\)](#).

---

**Note:** An efficient way to background-color a PDF page with the old Python paper color is

```
>>> col = fitz.utils.getColor("py_color")
>>> page.drawRect(page.rect, color=col, fill=col, overlay=False)
```

---

`insertFont(fontname="helv", fontfile=None, fontbuffer=None, set_simple=False)`

PDF only: Add a new font to be used by text output methods. 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. You have a “free” choice here, for a formal definition for valid PDF names see [Adobe PDF Reference 1.7](#) (page 56, section 3.2.4). 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` or `fontbuffer` and also give it a reserved `fontname`. The reserved font name can be specified in any mixture of upper or lower case and still match the right built-in font definition. When using `fontname` in outputting text however, the spelling chosen here must **exactly match**. The default value “helv” installs the built-in “Helvetica”.
- `fontfile` (*str*) – a path to a font file. If used, `fontname` must be different from all reserved names.
- `fontbuffer` (*bytes/bytearray*) – the 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.

**Rtype** `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 reader software is responsible for generating an appropriate

appearance – and their **are** differences on whether or how each one of them does this. This is especially true for the CJK fonts. Following are the reserved **Font Names** and their correspondingly installed **Base Font** names:

#### Base-14 Fonts<sup>51</sup>

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	
zadb	ZapfDingbats	

#### CJK Fonts<sup>52</sup>

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, overlay = True)`

PDF only: Fill the given rectangle (*rect*-like) with an image. The image's width-height-proportion will be **adjusted to fit** – so it may appear distorted. Specify the rectangle appropriately if you want to avoid this. The image is taken from a *pixmap*, a file or a memory area - of these parameters **exactly one** must be specified.

#### Parameters

- *rect* (*rect-like*) – where to put the image on the page. Must be finite and not empty.
- *filename* (*str*) – name of an image file (all MuPDF supported formats - see [Supported Input Image Types](#)). If the same image is to be inserted multiple times, choose one of the other two options to avoid some overhead.

---

<sup>51</sup> 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.

<sup>52</sup> Not all PDF reader software – which includes internet browsers and office software – will display all of these fonts. And if they do, the difference between the **serifed** and the **non-serifed** version may hardly be noticable. But serifed and non-serifed versions lead to different installed base fonts, thus providing an option to achieve desired results with a specific PDF reader.

- `stream(bytes/bytearray)` – memory resident image (all MuPDF supported formats - see [Supported Input Image Types](#)).
- `pixmap(Pixmap)` – pixmap containing 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(...)
```

Notes:

1. If that same image had already been present in the PDF, then only a reference will be inserted. This of course considerably saves disk space and processing time. But to detect this fact, existing PDF images need to be compared with the new one. This is achieved by storing an MD5 code for each image in a table and only compare the new image's code against the table entries. Generating this MD5 table, however, is done only when doing the first image insertion - 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 or a background color. Or you can combine it with `searchFor()` to achieve a textmarker effect. Please remember, that watermarks require a transparent image ...
3. The image may be inserted uncompressed, e.g. if a Pixmap is used or if the image has an alpha channel. Therefore, consider using `deflate = True` when saving the file.
4. The image content is stored in its original size - which may be much bigger than the size you are ever displaying. Consider decreasing the stored image size by using the `pixmap` option and then shrinking it or scaling it down (see [Pixmap](#) chapter). The file size savings can be very significant.
5. The most efficient way to display the same image on multiple pages is [showPDFpage\(\)](#). Consult [Document.convertToPDF\(\)](#) for how to obtain intermediary PDFs usable for that method. Demo script [fitz-logo.py](#)<sup>47</sup> implements a fairly complete approach.

`getText(output = 'text')`

Retrieves the content of a page in a large variety of formats.

If 'text' is specified, plain text is returned **in the order as specified during document creation** (i.e. not necessarily the normal reading order).

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

**Return type** (*str* or *dict*)

**Returns** The page's content as one string or as a dictionary. The information levels of JSON and DICT are exactly equal. In fact, JSON output is created via `json.dumps(...)` from DICT. Normally, you probably will use "dict", it is more convenient and faster.

<sup>47</sup> <https://github.com/rk700/PyMuPDF/blob/master/demo/fitz-logo.py>

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

---

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

`getFontList()`

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

`getImageList()`

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

`getSVGImage(matrix = fitz.Identity)`

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

**Parameters** `matrix` (*matrix-like*) – a matrix, default is `Identity`.

**Returns** a UTF-8 encoded string that contains the image. Because SVG has XML syntax it can be saved in a text file with extension `.svg`.

`getPixmap(matrix = fitz.Identity, colorspace = fitz.csRGB, clip = None, alpha = True)`

Create a pixmap from the page. This is probably the most often used method to create pixmaps.

**Parameters**

- `matrix` (*matrix-like*) – a matrix-like object, default is `Identity`.
- `colorspace` (string, [Colorspace](#)) – Defines the required colorspace, one of GRAY, RGB or CMYK (case insensitive). Or specify a [Colorspace](#), e.g. one of the predefined ones: `csGRAY`, `csRGB` or `csCMYK`.
- `clip` (*irect-like*) – restrict rendering to this area.
- `alpha` (*bool*) – A bool indicating whether an alpha channel should be included in the pixmap. Choose `False` if you do not really need transparency. This will save a lot of memory (25% in case of RGB ... and pixmaps are typically **large!**), and also processing time in most cases. Also note an important difference in how the image will appear:
  - `True`: pixmap's samples will be pre-cleared with `0x00`, including the alpha byte. This will result in **transparent** areas where the page is empty.



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



**Return type** *Pixmap*

**Returns** Pixmap of the page.

`loadLinks()`

Return the first link on a page. Synonym of property `firstLink`.

**Return type** *Link*

**Returns** first link on the page (or None).

`setRotation(rotate)`

PDF only: Sets the rotation of the page.

**Parameters** `rotate` (*int*) – An integer specifying the required rotation in degrees. Should be an integer multiple of 90.

`showPDFpage(rect, docsrc, pno = 0, keep_proportion = True, overlay = True, reuse_xref = 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, use it to

- create “n-up” versions of existing PDF files, combining several input pages into **one output page** (see example *4-up.py*<sup>48</sup>),
- create “posterized” PDF files, i.e. every input page is split up in parts which each create a separate output page (see *posterize.py*<sup>49</sup>),
- include PDF-based vector images like company logos, watermarks, etc., see *svg-logo.py*<sup>50</sup>, which puts an SVG-based logo on each page (requires additional packages to deal with SVG-to-PDF conversions).

<sup>48</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/4-up.py>

<sup>49</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/posterize.py>

<sup>50</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/svg-logo.py>

### Parameters

- `rect` (*rect-like*) – where to place the image.
- `docsrc` (*Document*) – source PDF document containing the page. Must be a different document object, but may be the same file.
- `pno` (*int*) – page number (0-based) to be shown.
- `keep_proportion` (*bool*) – whether to maintain the width-height-ratio (default).
- `overlay` (*bool*) – put image in foreground (default) or background.
- `reuse_xref` (*int*) – if a source page should be shown multiple times, specify the returned xref number of its first inclusion. This prevents duplicate source page copies, and thus improves performance and saves memory. Note that source document and page must still be provided!
- `clip` (*rect-like*) – choose which part of the source page to show. Default is its `/CropBox`.

**Returns** xref number of the stored page image if successful. Use this as the value of argument `reuse_xref` to show the same source page again.

---

**Note:** The displayed source page is shown without any annotations or links. The source page's complete text and images will become an integral part of the containing page, i.e. they will be included in the output of all text extraction methods and appear in methods `getFontList()` and `getImageList()` (whether they are actually visible - see the `clip` parameter - or not).

---

---

**Note:** Use the `reuse_xref` argument to prevent duplicates as follows. For a technical description of how this function is implemented, see [Design of Method Page.showPDFpage\(\)](#). The following example will put the same source page (probably a company logo or a watermark) on every page of PDF doc. The first execution **actually** inserts the source page, the subsequent ones will only insert pointers to it via its xref number.

---

```
>>> # the first showPDFpage will include source page docsrc[pno],
>>> # subsequents will reuse it via its xref.
>>> xref = 0
>>> for page in doc:
>>>     xref = page.showPDFpage(rect, docsrc, pno,
>>>                             reuse_xref = xref)
```

`newShape()`

PDF only: Create a new *Shape* object for the page.

**Return type** *Shape*

**Returns** a new *Shape* to use for compound drawings. See description there.

`searchFor(text, hit_max = 16, quads = False)`

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

### Parameters

- `text` (*str*) – Text to search for. Upper / lower case is ignored. The string may contain spaces.
- `hit_max` (*int*) – Maximum number of occurrences accepted.



- `quads (bool)` – Return *Quad* instead of *Rect* objects.

**Return type** list

**Returns** A list of rectangles (quadrilaterals resp.) each of which surrounds one occurrence of text.

`setCropBox(r)`

PDF only: change the visible part of the page.

**Parameters** `r (rect-like)` – the new visible area of the page.

After execution, `Page.rect` will equal this rectangle, shifted to the top-left position (0, 0). Example session:

```
>>> page = doc.newPage()
>>> page.rect
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> page.CropBox                # CropBox and MediaBox still equal
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> # now set CropBox to a part of the page
>>> page.setCropBox(fitz.Rect(100, 100, 400, 400))
>>> # this will also change the "rect" property:
>>> page.rect
fitz.Rect(0.0, 0.0, 300.0, 300.0)
>>>
>>> # but MediaBox remains unaffected
>>> page.MediaBox
fitz.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> # revert everything we did
>>> page.setCropBox(page.MediaBox)
>>> page.rect
fitz.Rect(0.0, 0.0, 595.0, 842.0)
```

`rotation`

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

**Type** int

`CropBoxPosition`

Contains the displacement of the page's `/CropBox` for a PDF, otherwise the top-left coordinates of `Page.rect`.

**Type** *Point*

`CropBox`

The page's `/CropBox` for a PDF, else `Page.rect`.

**Type** *Rect*

`MediaBoxSize`

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

**Type** *Point*

`MediaBox`

The page's `/MediaBox` for a PDF, otherwise `Page.rect`.

**Type** *Rect*

---

**Note:** For most PDF documents and for all other types, `page.rect == page.CropBox == page.MediaBox` is true. However, for some PDFs the visible page is a true subset of `/MediaBox`. In this case the above attributes help to correctly locate page elements.

---

`firstLink`

Contains the first [Link](#) of a page (or None).

**Type** [Link](#)

`firstAnnot`

Contains the first [Annot](#) of a page (or None).

**Type** [Annot](#)

`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 cross reference number. Zero if not a PDF.

**Type** [Rect](#)

---

### 4.3.2 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 cross reference entry of the link object. Do not change this entry in any way. Required for link deletion and update, otherwise ignored. For non-PDF documents, this entry contains -1. It is also -1 for **all** entries in the `getLinks()` list, if **any** of the links is not supported by MuPDF - see the note below.

### 4.3.3 Notes on Supporting Links

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

#### Reading (pertains to method `getLinks()` and the `firstLink` property chain)

If MuPDF detects a link to another file, it will supply either a `LINK_GOTOR` or a `LINK_LAUNCH` link kind. In case of `LINK_GOTOR` destination details may either be given as page number (eventually including position information), or as an indirect destination.

If an indirect destination is given, then this is indicated by `page = -1`, and `link.dest.dest` will contain this name. The dictionaries in the `getLinks()` list will contain this information as the `to` value.

**Internal links are always** of kind `LINK_GOTO`. If an internal link specifies an indirect destination, it **will always be resolved** and the resulting direct destination will be returned. Names are **never returned for internal links**, and undefined destinations will cause the link to be ignored.

#### Writing

PyMuPDF writes (updates, inserts) links by constructing and writing the appropriate PDF object **source**. This makes it possible to specify indirect destinations for `LINK_GOTOR` and `LINK_GOTO` link kinds (pre PDF 1.2 file formats are **not supported**).

**Caution:** If a `LINK_GOTO` indirect destination specifies an undefined name, this link can later on not be found / read again with MuPDF / PyMuPDF. Other readers however **will** detect it, but flag it as erroneous.

Indirect `LINK_GOTOR` destinations can in general of course not be checked for validity and are therefore **always accepted**.

### 4.3.4 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
<code>Document.getPageFontlist(pno)</code>	<code>Page.getFontList()</code>
<code>Document.getPageImageList(pno)</code>	<code>Page.getImageList()</code>
<code>Document.getPagePixmap(pno, ...)</code>	<code>Page.getPixmap()</code>
<code>Document.getPageText(pno, ...)</code>	<code>Page.getText()</code>
<code>Document.searchPageFor(pno, ...)</code>	<code>Page.searchFor()</code>

The page number `pno` is 0-based and can be any negative or positive number  $< \text{len}(\text{doc})$ .

**Technical Side 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)`.

## 4.4 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 Types](#).

---

Have a look at the [Pixmap Code Snippets](#) 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.getPNGData()</i>	return a PNG as a memory area
<i>Pixmap.invertIRect()</i>	invert the pixels of a given area
<i>Pixmap.setAlpha()</i>	sets alpha values
<i>Pixmap.shrink()</i>	reduce size keeping proportions
<i>Pixmap.tintWith()</i>	tint a pixmap with a color
<i>Pixmap.writeImage()</i>	save a pixmap in various formats
<i>Pixmap.writePNG()</i>	save a pixmap as a PNG file
<i>Pixmap.alpha</i>	transparency indicator
<i>Pixmap.colorspace</i>	pixmap's <i>Colorspace</i>
<i>Pixmap.height</i>	pixmap height
<i>Pixmap.interpolate</i>	interpolation method indicator
<i>Pixmap.irect</i>	<i>IRect</i> of the pixmap
<i>Pixmap.n</i>	bytes per pixel
<i>Pixmap.samples</i>	pixel area
<i>Pixmap.size</i>	pixmap's total length
<i>Pixmap.stride</i>	size of one image row
<i>Pixmap.width</i>	pixmap width
<i>Pixmap.x</i>	X-coordinate of top-left corner
<i>Pixmap.xres</i>	resolution in X-direction
<i>Pixmap.y</i>	Y-coordinate of top-left corner
<i>Pixmap.yres</i>	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 `clearWith()` to be sure.

### Parameters

- `colorspace` (*Colorspace*) – colorspace.
- `irect` (*IRect*) – Tte pixmap's position and dimension.
- `alpha` (*bool*) – Specifies whether transparency bytes should be included. Default is `False`.

`__init__(self, colorspace, source)`

**Copy and set colorspace:** Copy source pixmap converting colorspace. Any colorspace combination is possible, but source colorspace must not be `None`.

### Parameters

- `colorspace` (*Colorspace*) – desired target colorspace. This **may also be** `None`. In this case, a “masking” pixmap is created: its `Pixmap.samples` will consist of the source's alpha bytes only.
- `source` (Pixmap) – the source pixmap.

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

**Copy and scale:** Copy `source` pixmap choosing new width and height values. Supports partial copying and the source colorspace may be `None`.

**Parameters**

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

---

**Note:** If width or height are in fact no integers, the pixmap will be created with `alpha = 1`.

---

```
__init__(self, source, alpha = 1)
```

**Copy and add or drop alpha:** Copy `source` and add or drop its alpha channel. Identical copy if `alpha` equals `source.alpha`. If an alpha channel is added, its values will be set to 255.

**Parameters**

- `source` (`Pixmap`) – source pixmap.
- `alpha` (`bool`) – whether the target will have an alpha channel, default and mandatory if source colorspace is `None`.

```
__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, img)
```

**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** `img` (`bytes/bytearray`) – Data containing a complete, valid image. Could have been created by e.g. `img = bytearray(open('image.file', 'rb').read())`. Type `bytes` is supported in **Python 3 only**.

```
__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 [Pixmap Code Snippets](#) below.

**Parameters**

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

**Note:** The following equation **must be true**:  $(\text{colorspace.n} + \text{alpha}) * \text{width} * \text{height} == \text{len}(\text{samples})$ .

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

`__init__(self, doc, xref)`

**From a PDF image:** Create a pixmap from an image **contained in PDF** `doc` identified by its XREF number. All pixmap properties are set by the image. Have a look at [extract-img1.py](#)<sup>53</sup> and [extract-img2.py](#)<sup>54</sup> 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 number 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*) – 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. Pixmap 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  $2^n$ .

<sup>53</sup> <https://github.com/rk700/PyMuPDF/tree/master/demo/extract-img1.py>

<sup>54</sup> <https://github.com/rk700/PyMuPDF/tree/master/demo/extract-img2.py>

**Parameters** `n (int)` – determines the new pixmap (samples) size. For example, a value of 2 divides width and height by 4 and thus results in a size of one 16<sup>th</sup> 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.

---

`setAlpha([alphavalues])`

Change the alpha values. The pixmap must have an alpha channel.

**Parameters** `alphavalues (bytes/bytearray)` – 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).

`invertIRect([irect])`

Invert the color of all pixels in *IRect* `irect`. Will have no effect if colorspace is `None`.

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

`copyPixmap(source, irect)`

Copy the *IRect* part of `source` into the corresponding area of this one. The two pixmaps may have different dimensions and different colorspace (provided each is either *CS\_GRAY* or *CS\_RGB*), but currently **must** have the same alpha property. The copy mechanism automatically adjusts discrepancies between source and target like so:

If copying from *CS\_GRAY* to *CS\_RGB*, the source gray-shade value will be put into each of the three rgb component bytes. If the other way round,  $(r + g + b) / 3$  will be taken as the gray-shade value of the target.

Between `irect` and the target pixmap’s rectangle, an “intersection” is calculated at first. Then the corresponding data of this intersection are being copied. If this intersection is empty, nothing will happen.

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

**Parameters**

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

`writeImage(filename, output="png")`

Save pixmap as an image file. Depending on the output chosen, only some or all colorspace are supported and different file extensions can be chosen. Please see the table below. Since MuPDF v1.10a the `savealpha` option is no longer supported and will be ignored with a warning.

**Parameters**

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



`writePNG(filename)`

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

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

`getPNGData()`

Like `writePNG` but returns `bytearray` / `bytes` instead (Python2, resp. Python 3).

**Return type** `bytearray` / `bytes`

`alpha`

Indicates whether the pixmap contains transparency information.

**Type** `bool`

`colorspace`

The colorspace of the pixmap. This value may be `None` if the image is to be treated as a so-called *image mask* or *stencil mask* (currently happens for extracted PDF document images only).

**Type** *Colorspace*

`stride`

Contains the length of one row of image data in `samples`. This is primarily used for calculation purposes. The following expressions are true: `len(samples) == height * stride`, `width * n == stride`.

**Type** `int`

`irect`

Contains the *IRect* of the pixmap.

**Type** *IRect*

`samples`

The color and (if `alpha == 1`) transparency values for all pixels. `samples` is a memory area of size `width * height * n` bytes. Each `n` bytes define one pixel. Each successive `n` bytes yield another pixel in scanline order. Subsequent scanlines follow each other with no padding. E.g. for an RGBA colorspace this means, `samples` is a sequence of bytes like ..., R, G, B, A, ..., and the four byte values R, G, B, A define one pixel.

This area can be passed to other graphics libraries like PIL (Python Imaging Library) to do additional processing like saving the pixmap in other image formats. See example 3.

**Type** `bytes`

`size`

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

**Type** `int`

`width`

`w`

Width of the region in pixels.

**Type** `int`

`height`

<code>h</code>	Height of the region in pixels. <b>Type</b> <code>int</code>
<code>x</code>	X-coordinate of top-left corner <b>Type</b> <code>int</code>
<code>y</code>	Y-coordinate of top-left corner <b>Type</b> <code>int</code>
<code>n</code>	Number of components per pixel. This number depends on colorspace and alpha. If colorspace is not <code>None</code> (stencil masks), then <code>Pixmap.n - Pixmap.alpha == pixmap.colorsplace.n</code> is true. If colorspace is <code>None</code> , then <code>n == alpha == 1</code> . <b>Type</b> <code>int</code>
<code>xres</code>	Horizontal resolution in dpi (dots per inch). <b>Type</b> <code>int</code>
<code>yres</code>	Vertical resolution in dpi. <b>Type</b> <code>int</code>
<code>interpolate</code>	An information-only boolean flag set to <code>True</code> if the image will be drawn using “linear interpolation”. If <code>False</code> “nearest neighbour sampling” will be used. <b>Type</b> <code>bool</code>

#### 4.4.1 Supported Input Image Types

The following file types are supported as input to construct pixmaps: **BMP, JPEG, GIF, TIFF, JXR**, and **PNG**. This support is two-fold:

1. Directly create a pixmap with `Pixmap(filename)` or `Pixmap(byterray)`. The pixmap will then have properties as determined by the image.
2. Open such files with `fitz.open(...)`. The result will then appear as a document containing one single page. Creating a pixmap of this page offers all the options available in this context: apply a matrix, choose colorspace and alpha, confine the pixmap to a clip area, etc.

**SVG images** are only supported via method 2 above, not directly as pixmaps. But remember: the result of this is a **raster image** as is always the case with pixmaps.

If you need a **vector image** from the SVG, you must first convert it to a PDF. Try `Document.convertToPDF()`. If this does not work for you, look for other SVG-to-PDF conversion tools like the Python packages [svglib](https://pypi.org/project/svglib)<sup>55</sup>, [CairoSVG](https://pypi.org/project/cairosvg)<sup>56</sup>, [Uniconvertor](https://sk1project.net/modules.php?name=Products&product=uniconvertor&op=download)<sup>57</sup> or the Java solution [Apache Batik](https://github.com/apache/batik)<sup>58</sup>. Have a look at our Wiki for more examples.

---

<sup>55</sup> <https://pypi.org/project/svglib>

<sup>56</sup> <https://pypi.org/project/cairosvg>

<sup>57</sup> <https://sk1project.net/modules.php?name=Products&product=uniconvertor&op=download>

<sup>58</sup> <https://github.com/apache/batik>

#### 4.4.2 Details on Saving Images with `writeImage()`

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

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

**Note:** Not all image file types are available (or at least common) on all platforms. E.g. PAM is mostly unknown on Windows. Especially pertaining to CMYK colorspaces, you can always convert a CMYK pixmap to an RGB pixmap with `rgb_pix = fitz.Pixmap(fitz.csRGB, cmyk_pix)` and then save that as a PNG.

#### 4.4.3 Pixmap Code Snippets

##### Example 1: Glueing Images

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

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

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

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

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



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



### Example 2: Interfacing 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):
```

(continues on next page)

(continued from previous page)

```

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

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

```

### Example 3: Interfacing with PIL / Pillow

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

```

>>> import fitz
>>> from PIL import Image
>>> pix = fitz.Pixmap(..., alpha = False)
>>> ...
>>> # create and save a PIL image
>>> img = Image.frombytes("RGB", [pix.width, pix.height], pix.samples)
>>> img.save(filename, 'jpeg')
>>> ...
>>> # opposite direction:
>>> # create a pixmap from any PIL-supported image file "some_image.xxx"
>>> img = Image.open("some_image.xxx").convert("RGB")
>>> samples = img.tobytes()
>>> pix = fitz.Pixmap(fitz.csRGB, img.size[0], img.size[1], samples, alpha=False)

```

### Example 4: Extracting Alpha Values, Making Stencil Masks

The alpha channel of a pixmap can be extracted by making a copy and choosing target colorspace `None`. The resulting pixmap is sometimes also called “stencil mask”. Its samples contain the source’s alpha values.

```

>>> pix
fitz.Pixmap(DeviceRGB, fitz.IRect(0, 0, 1168, 823), 1)
>>> pix.n
4
>>> mask = fitz.Pixmap(None, pix)
>>> # now mask.samples will contain the alpha values of pix:
>>> mask
fitz.Pixmap(None, fitz.IRect(0, 0, 1168, 823), 1)
>>> mask.n
1

```

### Example 4: Back-Converting Stencil Masks

Stencil masks can be converted to PNG images. Use this to create a `DeviceGRAY` pixmap version:

```

>>> mask # stencil mask from previous example
fitz.Pixmap(None, fitz.IRect(0, 0, 1168, 823), 1)
>>> pix = fitz.Pixmap(mask.getPNGData())
>>> pix

```

(continues on next page)

(continued from previous page)

```
fitz.Pixmap(DeviceGRAY, fitz.IRect(0, 0, 1168, 823), 0)
>>> # if required, invert the gray values
>>> pix.invertIRect()
```

## 4.5 Colorspace

Represents the color space of a *Pixmap*.

### Class API

```
class Colorspace
```

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

**Parameters** *n* (*int*) – A number identifying the colorspace. Possible values are *CS\_RGB*, *CS\_GRAY* and *CS\_CMYK*.

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

**Type** *str*

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

**type** *int*

### Predefined Colorspaces

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

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

## 4.6 Link

Represents a pointer to somewhere (this document, other documents, the internet). Links exist per document page, and they are forward-chained to each other, starting from an initial link which is accessible by the *Page.firstLink* property.

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

Attribute	Short Description
<code>Link.setBorder()</code>	modify border 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>	XREF number of the entry

## Class API

class Link

`setBorder(border)`

PDF only: Change border width and dashing properties.

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

`colors`

Meaningful for PDF only: A dictionary of two lists of floats in range  $0 \leq \text{float} \leq 1$  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 Reference 1.7](#) page 217 for more details.
- *style* – 1-byte border style: S (Solid) = solid rectangle surrounding the annotation, D (Dashed) = dashed rectangle surrounding the link, the dash pattern is specified by the *dashes* entry, B (Beveled) = a simulated embossed rectangle that appears to be raised above the surface of the page, I (Inset) = a simulated engraved rectangle that appears to be recessed below the surface of the page, U (Underline) = a single line along the bottom of the annotation rectangle.

**Return type** dict

`rect`

The area that can be clicked in untransformed coordinates.

**Type** *Rect*

`isExternal`

A bool specifying whether the link target is outside of the current document.

**Type** bool

`uri`

A string specifying the link target. The meaning of this property should be evaluated in conjunction with property `isExternal`. The value may be `None`, in which case `isExternal == False`. If `uri` starts with `file://`, `mailto:`, or an internet resource name, `isExternal` is `True`. In all other cases `isExternal == False` and `uri` points to an internal location. In case of PDF documents, this should either be `#nnnn` to indicate a 1-based (!) page number `nnnn`, or a named location. The format varies for other document types, e.g. `uri = '../FixedDoc.fdoc#PG_2_LNK_1'` for page number 2 (1-based) in an XPS document.

**Type** str

`xref`

An integer specifying the PDF cross reference number. Zero if not a PDF.

**Type** int

`next`

The next link or `None`.

**Type** Link

`dest`

The link destination details object.

**Type** *linkDest*

## 4.7 linkDest

Class representing the *dest* property of an outline entry or a link. Describes the destination to which such entries point.

Attribute	Short Description
<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 Reference 1.7](#)). Standard actions provided are NextPage, PrevPage, FirstPage, and LastPage.

**Type** str

newWindow

If true, the destination should be launched in a new window.

**Type** bool

page

The page number (in this or the target document) this destination points to. Only set if *linkDest.kind* is *LINK\_GOTOR* or *LINK\_GOTO*. May be -1 if *linkDest.kind* is *LINK\_GOTOR*. In this case *linkDest.dest* contains the **name** of a destination in the target document.

**Type** int

rb

The bottom right [Point](#) of this destination.

**Type** [Point](#)

uri

The name of the URI this destination points to.

Type str

## 4.8 Matrix

Matrix is a row-major 3x3 matrix used by image transformations in MuPDF (which complies with the respective concepts laid down in the [Adobe PDF Reference 1.7](#)). With matrices you can manipulate the rendered image of a page in a variety of ways: (parts of) the page can be rotated, zoomed, flipped, sheared and shifted by setting some or all of just six float values.

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

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

Please note:

- the below methods are just convenience functions – everything they do, can also be achieved by directly manipulating the six numerical values
- all manipulations can be combined – you can construct a matrix that rotates **and** shears **and** scales **and** shifts, etc. in one go. If you however choose to do this, do have a look at the **remarks** further down or at the [Adobe PDF Reference 1.7](#).

Method / Attribute	Description
<i>Matrix.preRotate()</i>	perform a rotation
<i>Matrix.preScale()</i>	perform a scaling
<i>Matrix.preShear()</i>	perform a shearing (skewing)
<i>Matrix.preTranslate()</i>	perform a translation (shifting)
<i>Matrix.concat()</i>	perform a matrix multiplication
<i>Matrix.invert()</i>	calculate the inverted matrix
<i>Matrix.a</i>	zoom factor X direction
<i>Matrix.b</i>	shearing effect Y direction
<i>Matrix.c</i>	shearing effect X direction
<i>Matrix.d</i>	zoom factor Y direction
<i>Matrix.e</i>	horizontal shift
<i>Matrix.f</i>	vertical shift
<i>Matrix.isRectilinear</i>	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 modifiable versions of the *Identity* matrix, which looks like `[1, 0, 0, 1, 0, 0]`.

```
preRotate(deg)
```

Modify the matrix to perform a counter-clockwise rotation for positive `deg` degrees, else clockwise. The matrix elements of an identity matrix will change in the following way:

```
[1, 0, 0, 1, 0, 0] -> [cos(deg), sin(deg), -sin(deg), cos(deg), 0, 0].
```

**Parameters** `deg (float)` – The rotation angle in degrees (use conventional notation based on  $\text{Pi} = 180$  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

### 4.8.1 Remarks 1

This class adheres to the sequence protocol, so components can be maintained via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.

### 4.8.2 Remarks 2

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

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

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

### 4.8.3 Matrix Algebra

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

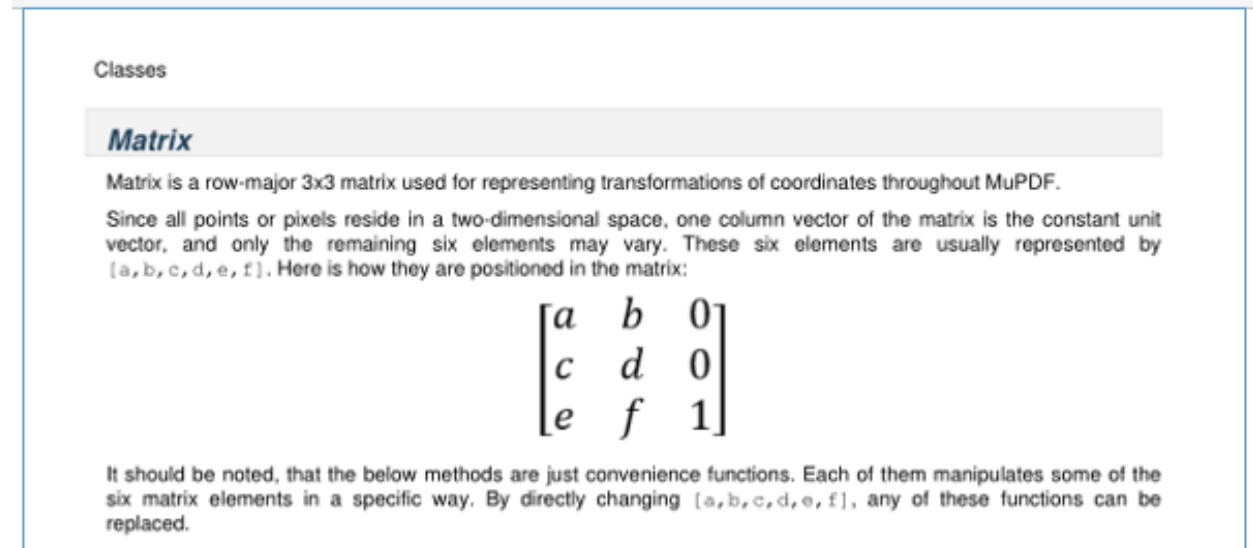
This makes the following operations possible:

```
>>> m45p = fitz.Matrix(45)           # rotate 45 degrees clockwise
>>> m45m = fitz.Matrix(-45)          # rotate 45 degrees counterclockwise
>>> m90p = fitz.Matrix(90)           # rotate 90 degrees clockwise
>>>
>>> abs(m45p * ~m45p - fitz.Identity) # should be (close to) zero:
8.429369702178807e-08
>>>
>>> abs(m90p - m45p * m45p)          # should be (close to) zero:
8.429369702178807e-08
>>>
>>> abs(m45p * m45m - fitz.Identity) # should be (close to) zero:
2.1073424255447017e-07
>>>
>>> abs(m45p - ~m45m)                # should be (close to) zero:
2.384185791015625e-07
>>>
>>> m90p * m90p * m90p * m90p        # should be 360 degrees = fitz.Identity
fitz.Matrix(1.0, -0.0, 0.0, 1.0, 0.0, 0.0)
```

#### 4.8.4 Examples

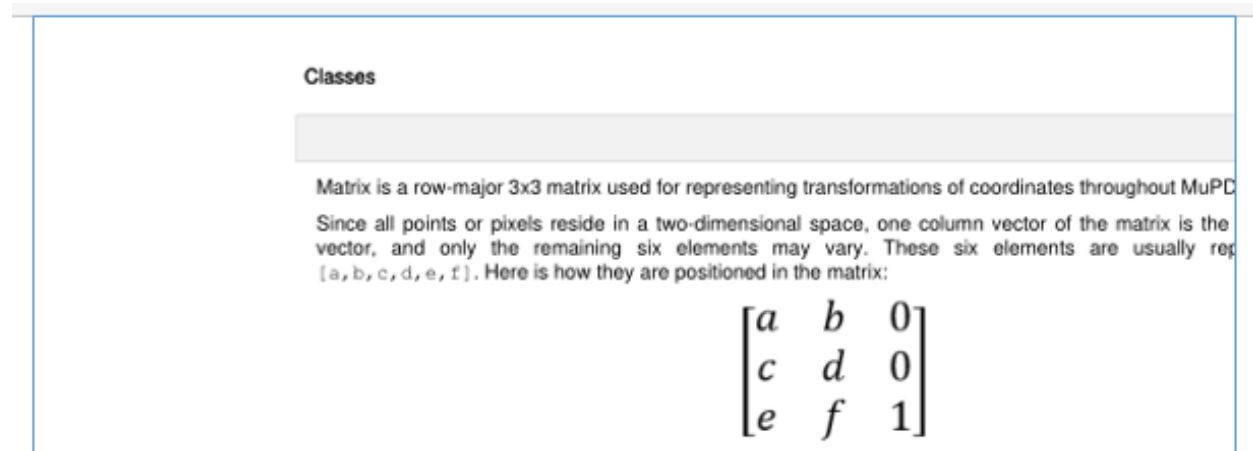
Here are examples to illustrate some of the effects achievable. The following pictures start with a page of the PDF version of this help file. We show what happens when a matrix is being applied (though always full pages are created, only parts are displayed here to save space).

This is the original page image:



#### 4.8.5 Shifting

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



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

## 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:

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

#### 4.8.6 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  $[a, b, c, d, e, f]$ . Here is how they are positioned in the matrix:

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

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

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

[a, b, c, d, e, f]. Here is how they are positioned in the matrix:

vector, and only the remaining six elements may vary. These six elements are usually represented by  $a, b, c, d, e, f$  since all points or pixels reside in a two-dimensional space, one column vector of the matrix is the constant unit vector.

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

## Matrix

Classes

### 4.8.7 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 combined - you can combine

## 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.

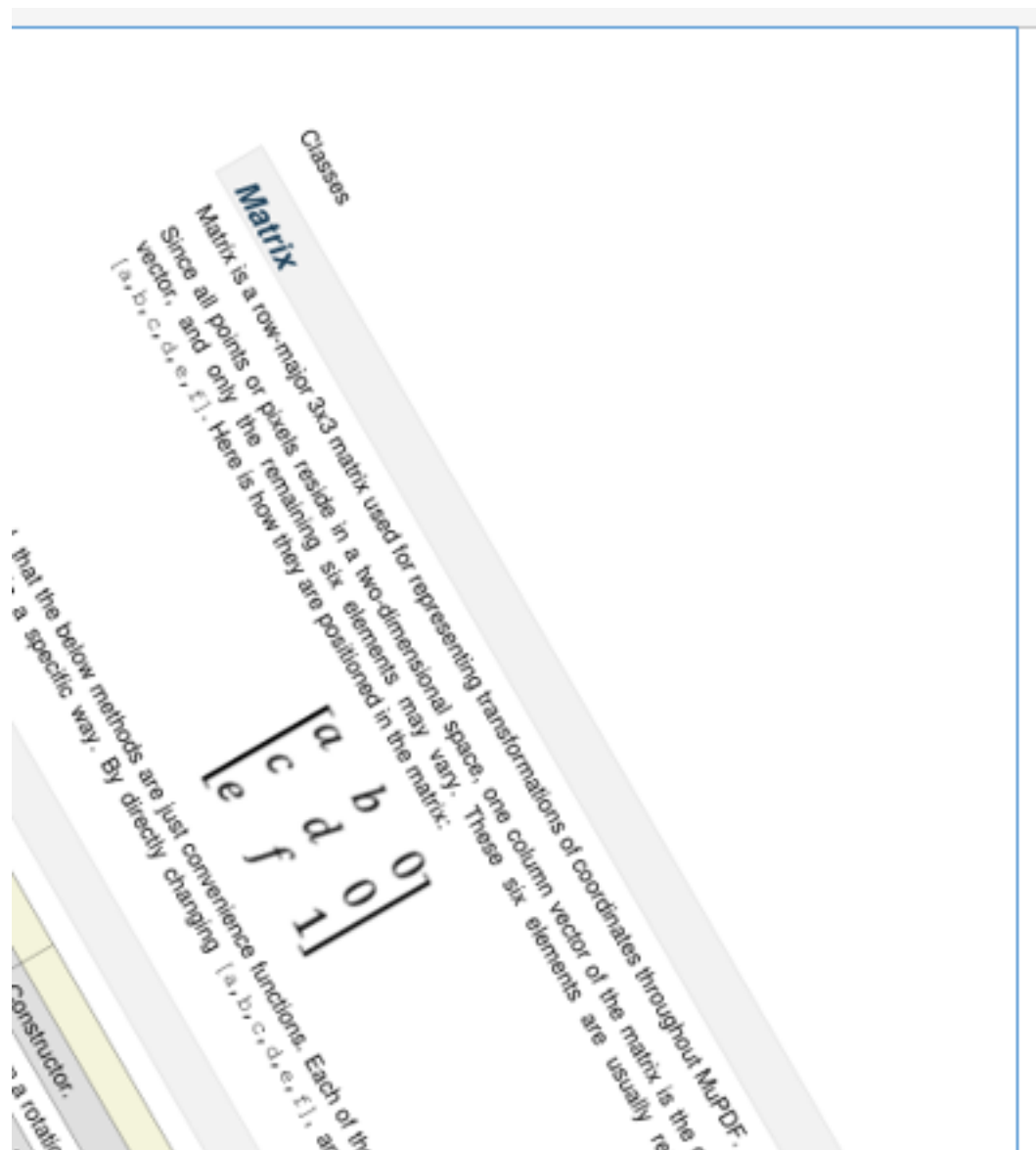
Since all points or pixels live in a two-dimensional space, one column vector of that matrix is a constant unit vector, and only the remaining six elements are used for manipulations. These six elements are usually represented by `[a,b,c,d,e,f]`. Here is how they are positioned in the matrix:

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

It should be noted, that

### 4.8.8 Rotating

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



## 4.9 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
```

## 4.10 IRect

IRect is a rectangular bounding box similar to [Rect](#), except that all corner coordinates are integers. IRect is used to specify an area of pixels, e.g. to receive image data during rendering. Otherwise, many similarities exist, e.g. considerations concerning emptiness and finiteness of rectangles also apply to IRects.

Attribute / Method	Short Description
<a href="#">IRect.contains()</a>	checks containment of another object
<a href="#">IRect.getArea()</a>	calculate rectangle area
<a href="#">IRect.getRect()</a>	return a <a href="#">Rect</a> with same coordinates
<a href="#">IRect.getRectArea()</a>	calculate rectangle area
<a href="#">IRect.intersect()</a>	common part with another rectangle
<a href="#">IRect.intersects()</a>	checks for non-empty intersection
<a href="#">IRect.normalize()</a>	makes a rectangle finite
<a href="#">IRect.bottom_left</a>	bottom left point, synonym bl
<a href="#">IRect.bottom_right</a>	bottom right point, synonym br
<a href="#">IRect.height</a>	height of the rectangle
<a href="#">IRect.isEmpty</a>	whether rectangle is empty
<a href="#">IRect.isInfinite</a>	whether rectangle is infinite
<a href="#">IRect.rect</a>	equals result of method <a href="#">getRect()</a>
<a href="#">IRect.top_left</a>	top left point, synonym tl
<a href="#">IRect.top_right</a>	top right point, synonym tr
<a href="#">IRect.quad</a>	<a href="#">Quad</a> made from rectangle corners
<a href="#">IRect.width</a>	width of the rectangle
<a href="#">IRect.x0</a>	X-coordinate of the top left corner
<a href="#">IRect.x1</a>	X-coordinate of the bottom right corner
<a href="#">IRect.y0</a>	Y-coordinate of the top left corner
<a href="#">IRect.y1</a>	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 integers (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 one of the rectangles is infinite, the other one is taken as the result – and hence also infinite if both rectangles were infinite.

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

`contains(x)`

Checks whether `x` is contained in the rectangle. It may be an *IRect*, *Rect*, “*Point*” or number. If `x` is an empty rectangle, this is always true. Conversely, if the rectangle is empty this is always `False`, if `x` is not an empty rectangle and not a number. If `x` is a number, it will be checked to be one of the four components. `x in irect` and `irect.contains(x)` are equivalent.

**Parameters** `x` (*IRect* or *Rect* or *Point* or `int`) – the object to check.

**Return type** `bool`

`intersects(r)`

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

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

**Return type** `bool`

`normalize()`

Make the rectangle finite. This is done by shuffling rectangle corners. After this, the bottom right corner will indeed be south-eastern to the top left one. See *Rect* for a more details.

`top_left`

`tl`

Equals `Point(x0, y0)`.

**Type** *Point*

`top_right`

`tr`

Equals `Point(x1, y0)`.

**Type** *Point*

`bottom_left`

`bl`

Equals `Point(x0, y1)`.

**Type** *Point*

`bottom_right`

`br`  
 Equals `Point(x1, y1)`.  
**Type** *Point*

`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`

#### 4.10.1 Remark

This class adheres to the sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.

#### 4.10.2 IRect Algebra

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

### 4.10.3 Examples

#### Example 1:

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

#### Example 2:

```
>>> m = fitz.Matrix(45)
>>> ir = fitz.Rect(10, 10, 410, 610)
>>> ir * m                # rotate rectangle by 45 degrees
fitz.Rect(-425, 14, 283, 722)
>>>
>>> ir | fitz.Point(5, 5)  # enlarge rectangle to contain a point
fitz.Rect(5, 5, 410, 610)
>>>
>>> ir + 5                # shift the rect by 5 points
fitz.Rect(15, 15, 415, 615)
>>>
>>> ir & fitz.Rect(0.0, 0.0, 15.0, 15.0)
fitz.Rect(10, 10, 15, 15)
>>> ir /= (1, 2, 3, 4, 5, 6) # divide by a matrix
>>> ir
fitz.Rect(-14, 0, 4, 8)
```

#### Example 3:

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

## 4.11 Rect

Rect represents a rectangle defined by four floating point numbers  $x_0$ ,  $y_0$ ,  $x_1$ ,  $y_1$ . They are viewed 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: `Rect(Px0,y0, Px1,y1)`, `Rect(Px1,y1, Px0,y0)`, `Rect(Px0,y1, Px1,y0)`, and `Rect(Px1,y0, Px0,y1)`.

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).
- 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.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 <code>bl</code>
<code>Rect.bottom_right</code>	bottom right point, synonym <code>br</code>
<code>Rect.height</code>	rectangle height
<code>Rect.irect</code>	equals result of method <code>round()</code>
<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 <code>tl</code>
<code>Rect.top_right</code>	top right point, synonym <code>tr</code>
<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* objects, “sequence” is a Python sequence type with 4 float values (see *Using Python Sequences as Arguments in PyMuPDF*), “rect” means another rectangle, while the other parameters mean float coordinates.

If “rect” is specified, the constructor creates a **new copy** of it.

Without parameters, the rectangle `Rect(0.0, 0.0, 0.0, 0.0)` is created.

`round()`

Creates the smallest containing *IRect* (this is **not** the same as simply rounding the rectangle’s edges!).

1. If the rectangle is **infinite**, the “normalized” (finite) version of it will be taken. The result of this method is always a finite *IRect*.
2. If the rectangle is **empty**, the result is also empty.
3. **Possible paradox:** The result may be empty, **even if** the rectangle is **not** empty! In such cases, the result obviously does **not** contain the rectangle. This is because MuPDF’s algorithm allows for a small tolerance (1e-3). Example:

```
>>> r = fitz.Rect(100, 100, 200, 100.001)
>>> r.isEmpty
False
>>> r.round()
fitz.IRect(100, 100, 200, 100)
>>> r.round().isEmpty
True
```

To reproduce this funny effect on your platform, you may need to adjust the numbers a little after the decimal point.

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

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

**Return type** `bool`

`normalize()`

**Replace** the rectangle with its finite version. This is done by shuffling the rectangle corners. After completion of this method, the bottom right corner will indeed be south-eastern to the top left one.

`irect`

Equals result of method `round()`.

`top_left`

`tl`

Equals `Point(x0, y0)`.

**Type** *Point*

`top_right`

`tr`

Equals `Point(x1, y0)`.

**Type** *Point*

`bottom_left`

`bl`

Equals `Point(x0, y1)`.

**Type** *Point*

`bottom_right`

`br`  
Equals `Point(x1, y1)`.  
**Type** *Point*

`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`

#### 4.11.1 Remark

This class adheres to the sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.

#### 4.11.2 Rect Algebra

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

### 4.11.3 Examples

#### Example 1 – different ways of construction:

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

#### Example 2 – what happens during rounding:

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

#### Example 3 – inclusion and iteration:

```
>>> m = fitz.Matrix(45)
>>> r = fitz.Rect(10, 10, 410, 610)
>>> r * m
fitz.Rect(-424.2640686035156, 14.142135620117188, 282.84271240234375, 721.2489013671875)
>>>
>>> r | fitz.Point(5, 5)
fitz.Rect(5.0, 5.0, 410.0, 610.0)
>>>
>>> r + 5
fitz.Rect(15.0, 15.0, 415.0, 615.0)
>>>
>>> r & fitz.Rect(0, 0, 15, 15)
fitz.Rect(10.0, 10.0, 15.0, 15.0)
```

#### Example 4 – containment:

```
>>> r = fitz.Rect(...)      # any rectangle
>>> ir = r.irect             # its IRect version
>>> # even though you get ...
>>> ir in r
True
>>> # ... and ...
>>> r in ir
True
>>> # ... r and ir are still different types!
>>> r == ir
False
```

(continues on next page)

(continued from previous page)

```
>>> # corners are always part of non-empty rectangles
>>> r.bottom_left in r
True
>>>
>>> # numbers are checked against coordinates
>>> r.x0 in r
True
```

**Example 5 – create a finite copy:**

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

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

**Example 6 – adding a Python sequence:**

Enlarge rectangle by 5 pixels in every direction:

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

**Example 7 – inline operations:**

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

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

## 4.12 Point

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

Attribute / Method	Description
<i>Point.distance_to()</i>	calculate distance to point or rect
<i>Point.transform()</i>	transform point with a matrix
<i>Point.abs_unit</i>	same as unit, but positive coordinates
<i>Point.unit</i>	point coordinates divides by abs(point)
<i>Point.x</i>	the X-coordinate
<i>Point.y</i>	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” must be Python sequence object of 2 floats (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])
```

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

---

**Note:** If `x` is a rectangle, the distance is calculated to the finite version of it.

---

#### Parameters

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

**Returns** distance to object `x`.

**Return type** float

```
transform(m)
```

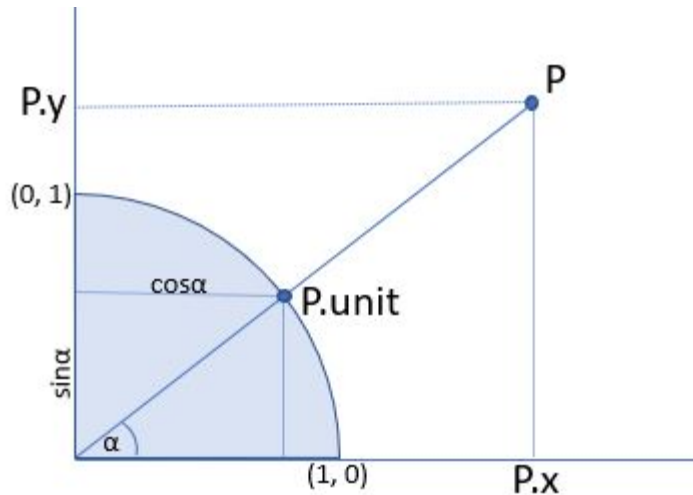
Applies matrix `m` to the point and replaces it with the result.

**Parameters** `m (Matrix)` – The matrix to be applied.

**Return type** [Point](#)

```
unit
```

Result of dividing each coordinate by `abs(point)`, which 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

#### 4.12.1 Remark

This class adheres to the sequence protocol, so components can be manipulated via their index. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.

#### 4.12.2 Point Algebra

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

#### 4.12.3 Examples

This should illustrate some basic uses:

```
>>> fitz.Point(1, 2) * fitz.Matrix(90)
fitz.Point(-2.0, 1.0)
>>>
>>> fitz.Point(1, 2) * 3
fitz.Point(3.0, 6.0)
>>>
>>> fitz.Point(1, 2) + 3
fitz.Point(4.0, 5.0)
```

(continues on next page)

(continued from previous page)

```

>>>
>>> fitz.Point(25, 30) + fitz.Point(1, 2)
fitz.Point(26.0, 32.0)
>>> fitz.Point(25, 30) + (1, 2)
fitz.Point(26.0, 32.0)
>>>
>>> fitz.Point([1, 2])
fitz.Point(1.0, 2.0)
>>>
>>> -fitz.Point(1, 2)
fitz.Point(-1.0, -2.0)
>>>
>>> abs(fitz.Point(25, 30))
39.05124837953327
>>>
>>> fitz.Point(1, 2) / (1, 2, 3, 4, 5, 6)
fitz.Point(2.0, -2.0)

```

## 4.13 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).

In (Py) MuPDF, only quads with four 90-degree angles and non-empty areas are of actual interest.

Such “interesting” 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).

---

**Note:** If *m* is a **rotation**, **scale** or a **translation Matrix**, and *rect* is a rectangle, then the four points *rect.tl* \* *m*, *rect.tr* \* *m*, *rect.bl* \* *m*, and *rect.br* \* *m* are the corners of a **rectangular quad**. This is **not in general true** – examples are shear matrices which produce parallelograms.

---



---

**Note:** This class provides an attribute to calculate the envelopping rectangle. Vice versa, rectangles now have the attribute *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.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.isEmpty</i>	true if corners define an empty area
<i>Quad.isRectangular</i>	true if all angles are 90 degrees
<i>Quad.rect</i>	smallest containing <a href="#">Rect</a>
<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](#) objects (the 4 corners), “sequence” is a Python sequence type with 4 [Point](#) objects.

If “quad” is specified, the constructor creates a **new copy** of it.

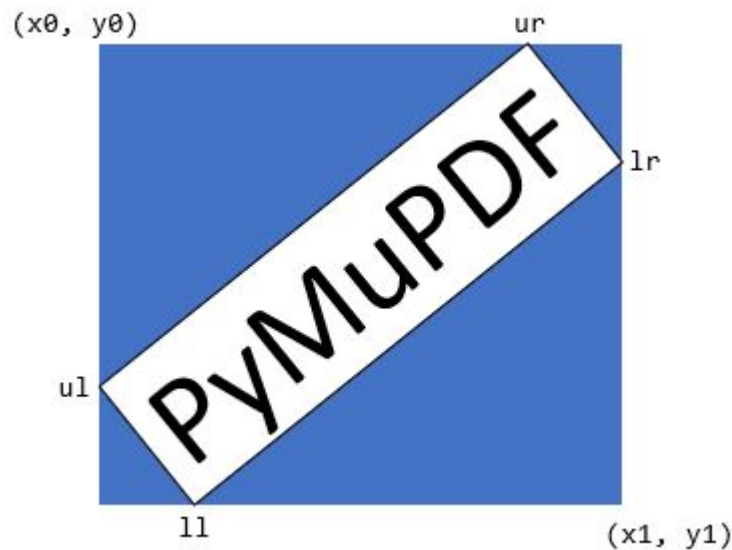
Without parameters, a quad consisting of 4 copies of `Point(0, 0)` is created.

```
transform(matrix)
```

Modify the quadrilateral by transforming each of its corners with a matrix.

```
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*

`isEmpty`  
True if enclosed area is zero, i.e. all points are on the same line. If this is false, the quad may still not look like a rectangle (but more like a triangle, trapezoid, etc.).  
**Type** `bool`

`isRectangular`  
True if all angles are 90 degrees. This also implies that the area is **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`

#### 4.13.1 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.

## 4.14 Shape

This class allows creating interconnected graphical elements on a PDF page. Its methods have the same meaning and name as the corresponding *Page* methods. Their *Common Parameters* are however exported to a separate method, `finish()`. In addition, all draw methods return a *Point* object to support connected drawing paths. This point always equals the “current point”, that PDF maintains during path construction.

Methods of this class record the area they are covering in a rectangle (*Shape.rect*). This property can for instance be used to set *Page.CropBox*.

Also supported are text insertion methods `insertText()` and `insertTextbox()`. They need a slightly different handling compared to the draw methods:

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

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

Method / Attribute	Description
<i>Shape.commit()</i>	update the page's /Contents object
<i>Shape.drawBezier()</i>	draw a cubic Bézier curve
<i>Shape.drawCircle()</i>	draw a circle around a point
<i>Shape.drawCurve()</i>	draw a cubic Bézier 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.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 draws
<i>Shape.insertText()</i>	insert text lines
<i>Shape.insertTextbox()</i>	insert text into a rectangle
<i>Shape.contents</i>	draw commands since last finish()
<i>Shape.doc</i>	stores the page's document
<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.width</i>	stores the page's width
<i>Shape.totalcont</i>	accumulated string to be stored in /Contents

## Class API

class Shape

`__init__(self, page)`

Create a new drawing. During importing PyMuPDF, the `fitz.Page` object is being given the convenience method `newShape()` to construct a `Shape` object. During instantiation, a check will be made whether we do have a PDF page. An exception is otherwise raised.

**Parameters** `page` (*Page*) – an existing page of a PDF document.

`drawLine(p1, p2)`

Draw a line from point-like objects `p1` to `p2`.

**Parameters**

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

**Return type** *Point*

**Returns** the end point, `p2`.

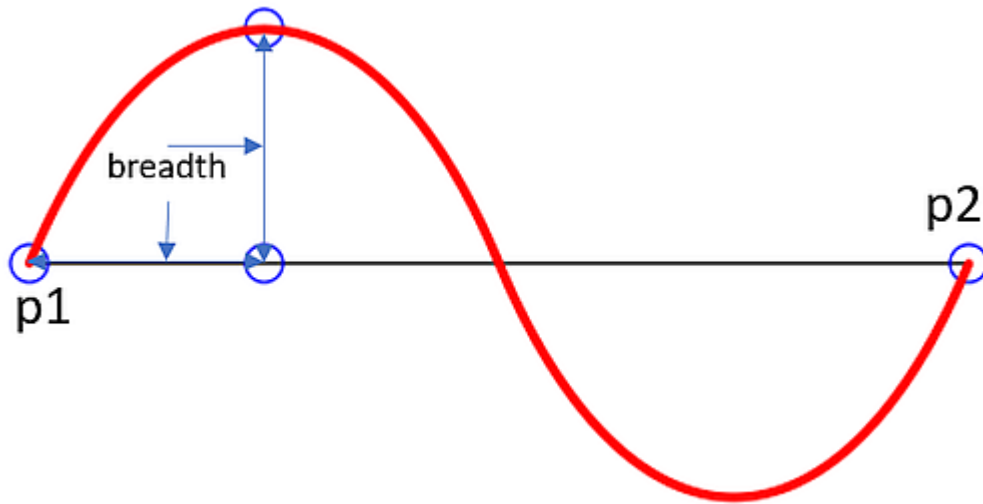
**..index::** pair: breadth; Shape.drawSquiggle args

`drawSquiggle(p1, p2, breadth = 2)`

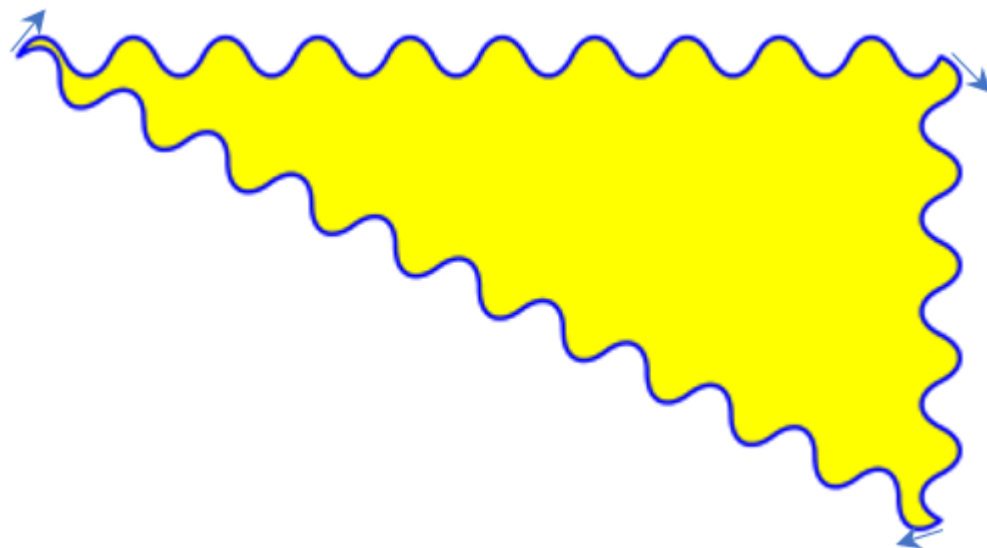
Draw a squiggly (wavy, undulated) line from point-like objects `p1` to `p2`. An integer number of full wave periods will always be drawn, one period having a length of `4 * breadth`. The `breadth` parameter will be adjusted as necessary to meet this condition. The drawn line will always turn “left” when leaving `p1` and always join `p2` from the “right”.

**Parameters**

- `p1` (*point-like*) – starting point
- `p2` (*point-like*) – end point
- `breadth` (*float*) – the amplitude of each wave. The condition  $2 * \text{breadth} < \text{abs}(p2 - p1)$  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](#)<sup>59</sup>.

---

**..index::** pair: breadth; Shape.drawZigzag args

`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  $4 * \text{breadth}$ . 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  $2 * \text{breadth} < \text{abs}(p2 - p1)$  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 Bézier curve from p1 to p4, using p2 and p3 as control points.

**Parameters**

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

**Return type** *Point*

**Returns** the end point, p4.

---

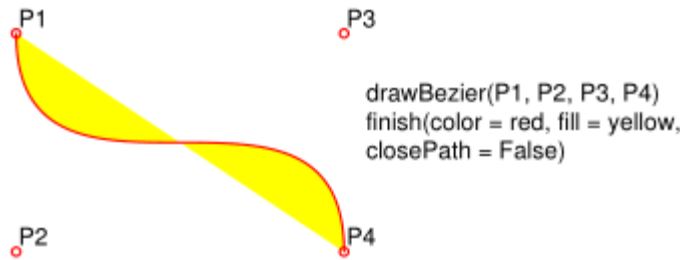
**Note:** The points do not need to be different – experiment a bit with some of them being equal!

---

Example:

---

<sup>59</sup> <https://github.com/rk700/PyMuPDF/blob/master/demo/draw-sines.py>



`drawOval(rect)`

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

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

**Return type** *Point*

**Returns** the middle point of the left rectangle side.

`drawCircle(center, radius)`

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

The method is a shortcut for `drawSector(center, start, 360, fullSector = False)`. To draw a circle in a clockwise movement, change the sign of the degree.

**Parameters**

- `center` (*point-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 Bézier curve from `p1` to `p3`. On each of the two lines from `p1` to `p2` and from `p2` to `p3` one control point is generated. This guaranties that the curve's curvature does not change its sign. If these two connecting lines intersect with an angle of 90 degrees, then the resulting curve is a quarter ellipse (or quarter circle, if of same length) circumference.

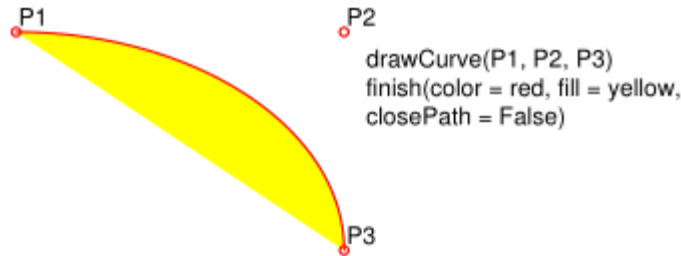
**Parameters**

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

**Return type** *Point*

**Returns** the end point, `p3`.

Example: a filled quarter ellipse segment.



`drawSector(center, point, angle, fullSector = True)`

Draw a circular sector, optionally connecting the arc to the circle's center (like a piece of pie).

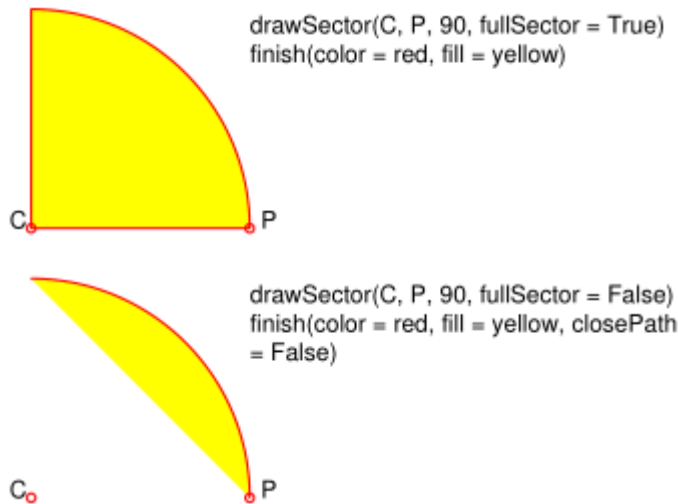
**Parameters**

- *center* (*point-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 counter-clockwise (positive) or clockwise.
- *fullSector* (*bool*) – whether to draw connecting lines from the ends of the arc to the circle center. If a fill color is specified, the full “pie” is colored, otherwise just the sector.

**Returns** the other end point of the arc. Can be used as starting point for a following invocation to create logically connected pies charts.

**Return type** *Point*

Examples:



`drawRect(rect)`

Draw a rectangle. The drawing starts and ends at the top-left corner in a counter-clockwise movement.

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

**Return type** *Point*

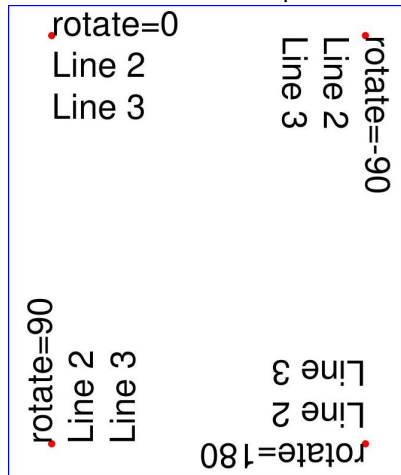
**Returns** `rect.top_left` (top-left corner of the rectangle).

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

Insert text lines start at point.

#### Parameters

- `point` (*point-like*) – the bottom-left position of the first character of `text` in pixels. It is important to understand, how this works in conjunction with the `rotate` parameter. Please have a look at the following picture. The small red dots indicate the positions of point in each of the four possible cases.



- `text` (*str/sequence*) – the text to be inserted. May be specified as either a string type or as a sequence type. For sequences, or strings containing line breaks `\n`, several lines will be inserted. No care will be taken if lines are too wide, but the number of inserted lines will be limited by “vertical” space on the page (in the sense of reading direction as established by the `rotate` parameter). Any rest of `text` is discarded – the return code however contains the number of inserted lines.
- `rotate` (*int*) – determines whether to rotate the text. Acceptable values are multiples of 90 degrees. Default is 0 (no rotation), meaning horizontal text lines oriented from left to right. 180 means text is shown upside down from **right to left**. 90 means counter-clockwise rotation, text running **upwards**. 270 (or -90) means clockwise rotation, text running **downwards**. In any case, `point` specifies the bottom-left coordinates of the first character’s rectangle. Multiple lines, if present, always follow the reading direction established by this parameter. So line 2 is located **above** line 1 in case of `rotate = 180`, etc.

**Return type** `int`

**Returns** number of lines inserted.

For a description of the other parameters see [Common Parameters](#).

```
insertTextbox(rect, buffer, fontsize = 11, fontname = "helv", fontfile = None, idx = 0, set_simple
              = False, color = (0, 0, 0), expandtabs = 8, align = TEXT_ALIGN_LEFT, rotate =
              0, morph = None)
```

PDF only: Insert text into the specified rectangle. The text will be split into lines and words and then filled into the available space, starting from one of the four rectangle corners, depending on `rotate`. Line feeds will be respected as well as multiple spaces will be.

#### Parameters

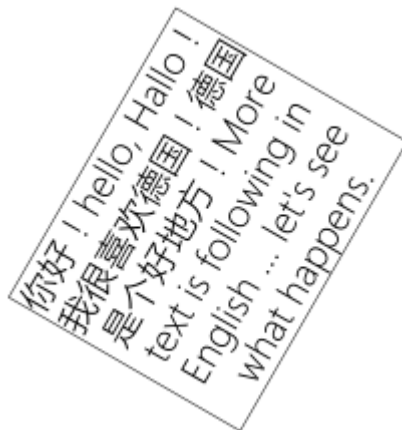
- `rect` (*rect-like*) – the area to use. It must be finite and not empty.
- `buffer` (*str/sequence*) – the text to be inserted. Must be specified as a string or a sequence of strings. Line breaks are respected also when occurring in a sequence entry.
- `align` (*int*) – align each text line. Default is 0 (left). Centered, right and justified are the other supported options, see [Text Alignment](#). Please note that the effect of parameter value `TEXT_ALIGN_JUSTIFY` is only achievable with “simple” (single-byte) fonts (including the [PDF Base 14 Fonts](#)). Refer to [Adobe PDF Reference 1.7](#), section 5.2.2, page 399.
- `expandtabs` (*int*) – controls handling of tab characters `\t` using the `string.expandtabs()` method **per each line**.
- `rotate` (*int*) – requests text to be rotated in the rectangle. This value must be a multiple of 90 degrees. Default is 0 (no rotation). Effectively, four different values are processed: 0, 90, 180 and 270 (= -90), each causing the text to start in a different rectangle corner. Bottom-left is 90, bottom-right is 180, and -90 / 270 is top-right. See the example how text is filled in a rectangle. This argument takes precedence over `morphing`. See the second example, which shows text first rotated left by 90 degrees and then the whole rectangle rotated clockwise around its lower left corner.

**Return type** float

#### Returns

**If positive or zero:** successful execution. The value returned is the unused rectangle line space in pixels. This may safely be ignored – or be used to optimize the rectangle, position subsequent items, etc.

**If negative:** no execution. The value returned is the space deficit to store text lines. Enlarge rectangle, decrease `fontsize`, decrease text amount, etc.



For a description of the other parameters see [Common Parameters](#).

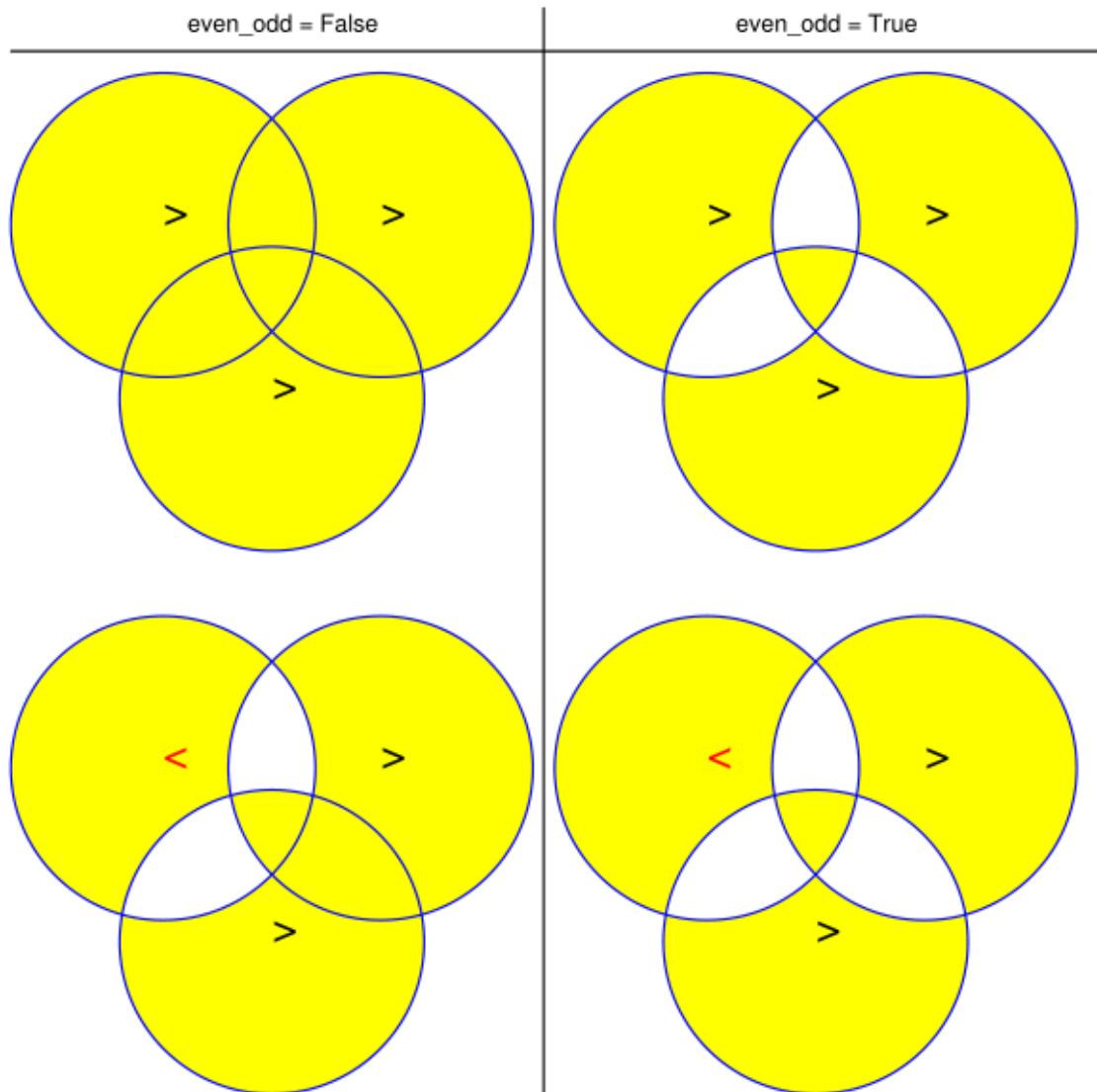


```
finish(width = 1, color = (0, 0, 0), fill = None, roundCap = True, dashes = None, closePath =  
      True, even_odd = False, morph = (pivot, matrix))
```

Finish a set of `draw*()` methods by applying [Common Parameters](#) to all of them. This method also supports morphing the resulting compound drawing using a pivotal [Point](#).

#### Parameters

- `morph(sequence)` – morph the compound drawing around some arbitrary pivotal [Point](#) pivot by applying [Matrix](#) matrix to it. Default is no morphing (`None`). The matrix can contain any values in its first 4 components, `matrix.e == matrix.f == 0` must be true, however. This means that any combination of scaling, shearing, rotating, flipping, etc. is possible, but translations are not.
- `even_odd(bool)` – request the “**even-odd rule**” for filling operations. Default is `False`, so that the “**nonzero winding number rule**” is used. These rules are alternative methods to apply the fill color where areas overlap. Only with fairly complex shapes a different behavior is to be expected with these rules. For an in-depth explanation, see [Adobe PDF Reference 1.7](#), pp. 232 ff. Here is an example to demonstrate the difference.



**Note:** Method “**even-odd**” counts the number of overlaps of areas. Pixels in areas overlapping an odd number of times are regarded **inside**, otherwise **outside**. In contrast, the default method “**nonzero winding**” also looks at the area orientation: it counts +1 if an area is drawn counter-clockwise and -1 else. If the result is zero, the pixel is regarded **outside**, otherwise **inside**. In the top two shapes, three circles are drawn in standard manner (anti-clockwise, look at the arrows). The lower two shapes contain one (top-left) circle drawn clockwise. As can be seen, area orientation is irrelevant for the even-odd rule.

`commit(overlay = True)`

Update the page’s `/Contents` with the accumulated drawing commands. If a `Shape` is not committed, the page will not be changed. The method must be preceded with at least one `finish()` or one text insertion method.

The method will reset attributes `Shape.rect`, `Shape.lastPoint`, `Shape.contents` and `Shape.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 has a non-empty `/Contents` object.

`doc`

For reference only: the page's document.

**Type** *Document*

`page`

For reference only: the owning page.

**Type** *Page*

`height`

Copy of the page's height

**Type** float

`width`

Copy of the page's width.

**Type** float

`contents`

Accumulated command buffer for draw methods since last finish.

**Type** str

`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 want to adjust this attribute accordingly, use the following code:

```
>>> # assuming ...
>>> morph = (point, matrix)
>>> # ... recalculate the shape rectangle like so:
>>> img.rect = (img.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*

### 4.14.1 Usage

A drawing object is constructed by `img = page.newShape()`. After this, as many `draw`, `finish` and `text` insertions methods as required may follow. Each sequence of draws must be finished before the drawing is committed. The overall coding pattern looks like this:

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

#### Notes

1. Each `finish()` combines the preceding draws into one logical shape, giving it common colors, line width, morphing, etc. If `closePath` is specified, it will also connect the end point of the last draw with the starting point of the first one.
2. To successfully create compound graphics, let each draw method use the end point of the previous one as its starting point. In the above pseudo code, `draw2` should hence use the returned *Point* of `draw1` as its starting point. Failing to do so, would automatically start a new path and `finish()` may not work as expected (but it won't complain either).
3. Text insertions may occur anywhere before the commit (they neither touch *Shape.contents* nor *Shape.lastPoint*). They are appended to *Shape.totalcont* directly, whereas draws will be appended by *Shape.finish*.
4. Each `commit` takes all text insertions and shapes and places them in foreground or background on the page – thus providing a way to control graphical layers.
5. Only `commit` will update the page's contents, the other methods are basically string manipulations. With many draw / text operations, this will result in a much better performance, than issuing the corresponding page methods separately (they each do their own commit).

### 4.14.2 Examples

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

```
>>> img = page.newShape()           # start a new shape
>>> cols = (...)                     # a sequence of RGB color triples
>>> pieces = len(cols)               # number of pieces to draw
>>> beta = 360. / pieces              # angle of each piece of pie
>>> center = fitz.Point(...)          # center of the pie
>>> p0 = fitz.Point(...)              # starting point
>>> for i in range(pieces):
    p0 = img.drawSector(center, p0, beta,
```

(continues on next page)

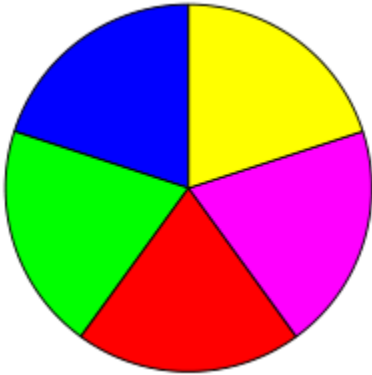
(continued from previous page)

```

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

```

Here is an example for 5 colors:



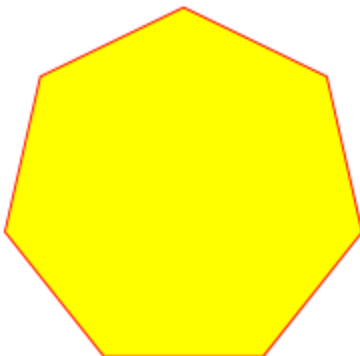
2. Create a regular n-edged polygon (fill yellow, red border). We use `drawSector()` only to calculate the points on the circumference, and empty the draw command buffer before drawing the polygon:

```

>>> img = page.newShape()          # start a new shape
>>> beta = -360.0 / n              # our angle, drawn clockwise
>>> center = fitz.Point(...)       # center of circle
>>> p0 = fitz.Point(...)           # start here (1st edge)
>>> points = [p0]                 # store polygon edges
>>> for i in range(n):             # calculate the edges
>>>     p0 = img.drawSector(center, p0, beta)
>>>     points.append(p0)
>>> img.contents = ""              # do not draw the circle sectors
>>> img.drawPolyline(points)        # draw the polygon
>>> img.finish(color = (1,0,0), fill = (1,1,0), closePath = False)
>>> img.commit()

```

Here is the polygon for  $n = 7$ :



### 4.14.3 Common Parameters

**fontname** (*str*)

In general, there are three options:

1. Use one of the standard [PDF Base 14 Fonts](#). In this case, `fontfile` **must not** be specified and "Helvetica" is used if this parameter is omitted, too.
2. Choose a font already in use by the page. Then specify its **reference** name prefixed with a slash "/", see example below.
3. Specify a font file present on your system. In this case choose an arbitrary, but new name for this parameter (without "/" prefix).

If inserted text should re-use one of the page's fonts, use its reference name appearing in `getFontList()` like so:

Suppose the font list has the entry `[1024, 0, 'Type1', 'CJXQIC+NimbusMonL-Bold', 'R366']`, then specify `fontname = "/R366"`, `fontfile = None` to use font `CJXQIC+NimbusMonL-Bold`.

---

#### **fontfile** (*str*)

File path of a font existing on your computer. If you specify `fontfile`, make sure you use a `fontname` **not occurring** in the above list. This new font will be embedded in the PDF upon `doc.save()`. Similar to new images, a font file will be embedded only once. A table of MD5 codes for the binary font contents is used to ensure this.

---

#### **idx** (*int*)

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

---

#### **set\_simple** (*bool*)

Fonts installed from files are installed as **Type0** fonts by default. If you want to use 1-byte characters only, set this to true. This setting cannot be reverted. Subsequent changes are ignored.

---

#### **fontsize** (*float*)

Font size of text. This also determines the line height as `fontsize * 1.2`.

---

#### **dashes** (*str*)

Causes lines to be dashed. A continuous line with no dashes is drawn with `[]0` or `None`. For (the rather complex) details on how to achieve dashing effects, see [Adobe PDF Reference 1.7](#), page 217. Simple versions look like `"[3 4]"`, which means dashes of 3 and gaps of 4 pixels length follow each other. `"[3 3]"` and `"[3]"` do the same thing.

---

#### **color / fill** (*list, tuple*)

Line and fill colors are always specified as RGB triples of floats from 0 to 1. To simplify color specification, method `getColor()` in `fitz.utils` may be used. It accepts a string as the name of the color and returns the corresponding triple. The method knows over 540 color names – see section [Color Database](#).

---

**overlay** (*bool*)

Causes the item to appear in foreground (default) or background.

**morph** (*sequence*)

Causes “morphing” of either a shape, created by the `draw*()` methods, or the text inserted by page methods `insertTextbox()` / `insertText()`. If not `None`, it must be a pair (`pivot`, `matrix`), where `pivot` is a [Point](#) and `matrix` is a [Matrix](#). The matrix can be anything except translations, i.e. `matrix.e == matrix.f == 0` must be true. The point is used as a pivotal point for the matrix operation. For example, if `matrix` is a rotation or scaling operation, then `pivot` is its center. Similarly, if `matrix` is a left-right or up-down flip, then the mirroring axis will be the vertical, respectively horizontal line going through `pivot`, etc.

**Note:** Several methods contain checks whether the to be inserted items will actually fit into the page (like `Shape.insertText()`, or `Shape.drawRect()`). For the result of a morphing operation there is however no such guaranty: this is entirely the programmer’s responsibility.

**roundCap** (*bool*)

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

**closePath** (*bool*)

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

## 4.15 Annot

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

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

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

Attribute	Short Description
<code>Annot.fileGet()</code>	PDF only: returns attached file content
<code>Annot.fileInfo()</code>	PDF only: returns attached file information
<code>Annot.fileUpd()</code>	PDF only: sets attached file new content

Continued on next page

Table 3 – continued from previous page

Attribute	Short Description
<code>Annot.getPixmap()</code>	image of the annotation as a pixmap
<code>Annot.setBorder()</code>	PDF only: changes the border of an annotation
<code>Annot.setColors()</code>	PDF only: changes the colors of an annotation
<code>Annot.setFlags()</code>	PDF only: changes the flags of an annotation
<code>Annot.setInfo()</code>	PDF only: change metadata of an annotation
<code>Annot.setLineEnds()</code>	PDF only: sets the line ending styles
<code>Annot.setOpacity()</code>	PDF only: changes the annot's transparency
<code>Annot.setRect()</code>	PDF only: changes the rectangle of an annotation
<code>Annot.update()</code>	PDF only: applies accumulated annot changes
<code>Annot.updateWidget()</code>	PDF only: change an existing form field
<code>Annot.border</code>	PDF only: border details
<code>Annot.colors</code>	PDF only: border / background and fill colors
<code>Annot.flags</code>	PDF only: annotation flags
<code>Annot.info</code>	PDF only: various information
<code>Annot.lineEnds</code>	PDF only: 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.rect</code>	rectangle containing the annotation
<code>Annot.type</code>	PDF only: type of the annotation
<code>Annot.vertices</code>	PDF only: point coordinates of Polygons, PolyLines, etc.
<code>Annot.widget</code>	PDF only: <a href="#">Widget</a> object for form fields
<code>Annot.widget_choices</code>	PDF only: possible values for "Widget" list / combo boxes
<code>Annot.widget_name</code>	PDF only: "Widget" field name
<code>Annot.widget_type</code>	PDF only: "Widget" field type
<code>Annot.widget_value</code>	PDF only: "Widget" field value
<code>Annot.xref</code>	the PDF XREF number

**Class API**

```
class Annot
```

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

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

**Parameters**

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

**Return type** [Pixmap](#)

```
setInfo(d)
```

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

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



`setLineEnds(start, end)`

PDF only: Sets an annotation's line ending styles. Only 'FreeText', 'Line', 'PolyLine', and 'Polygon' annotations can have these properties. Each of these annotation types is defined by a list of points which are connected by lines. The symbol identified by `start` is attached to the first point, and `end` to the last point of this list. For unsupported annotation types, a no-operation with a warning message results. See [Annotation Line End Styles](#) for details.

**Parameters**

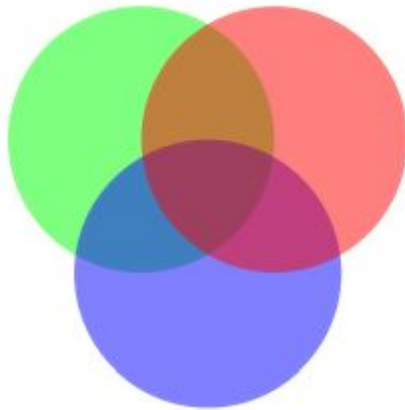
- `start (int)` – The symbol number for the first point.
- `end (int)` – The symbol number for the last point.

`setOpacity(value)`

PDF only: Change an annotation's transparency.

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

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



`setRect(rect)`

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

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

`setBorder(border)`

PDF only: Change border width and dashing properties.

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

`setFlags(flags)`

Changes the annotation flags. See [Annotation Flags](#) for possible values and use the | operator to combine several.

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

`setColors(d)`

PDF only: Changes the "stroke" and "fill" colors for supported annotation types.

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

---

**Note:** This method **does not work** for widget annotations, and results in a no-op with a warning message. Use `updateWidget()` instead. Certain annotation types have no fill colors. In these cases this value is ignored and a warning is issued. FreeText annotations also require a special handling – see `update()`.

---

`update(fontsize = 0, text_color = None, border_color = None, fill_color = None, rotate = -1)`

PDF only: Modify the displayed annotation image such that it coincides with the values contained in the `width`, `border`, `colors` and other properties, after they have been modified by the respective methods (like. `setBorder()`, `setColors()`, etc.).

It is ignored for widget annotations (use `updateWidget()` instead).

Without invoking this method, changes to any of these will be lost. All arguments are optional and **only work for FreeText** annotations – because of the peculiarities of how this annotation type is implemented by MuPDF. For other types they are ignored. Color specifications may be made in the usual format used in PuMuPDF as sequences of floats ranging from 0.0 to 1.0 (including both). The sequence length must be 1, 3 or 4 (supporting GRAY, RGB and CMYK colorspace respectively).

**Parameters**

- `fontsize` (*float*) – change font size of the text.
- `text_color` (*sequence*) – change the text color.
- `border_color` (*sequence*) – change the border color.
- `fill_color` (*sequence*) – the fill color.
- `rotate` (*int*) – new rotation value. Default (-1) means no change.

**Return type** `bool`

**Returns** `True` on success, else `False` (or `None` for non-PDFs).

`updateWidget(widget)`

Modifies an existing form field. The existing and the changed widget attributes must all be provided by way of a `Widget` object. This is because the method will update the field with **all properties** of the `Widget` object.

**Parameters** *widget* (`Widget`) – a widget object containing the **complete** (old and new) properties of the widget. Create this object by copying the `Annot.widget` attribute and applying your changes to it and then passing it to this method.

`fileInfo()`

Basic information of the annot's attached file.

**Return type** `dict`

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

`fileGet()`

Returns attached file content.

**Return type** `bytes`

**Returns** the content of the attached file.

`fileUpd(buffer = None, filename=None, ufilename=None, desc = None)`  
 Updates the content of an attached file.

**Parameters**

- `buffer (bytes/bytearray)` – the new file content. May be omitted to only change meta-information.
- `filename (str)` – new filename to associate with the file.
- `ufilename (str)` – new unicode filename to associate with the file.
- `desc (str)` – new description of the file content.

`opacity`

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

**Return type** float

`parent`

The owning page object of the annotation.

**Return type** [Page](#)

`rect`

The rectangle containing the annotation.

**Return type** [Rect](#)

`next`

The next annotation on this page or `None`.

**Return type** `Annot`

`type`

Meaningful for PDF only: A number and one or two strings describing the annotation type, like [2, 'FreeText', 'FreeTextCallout']. The second string entry is optional and may be empty. [] if not PDF. See the appendix [Annotation Types](#) for a list of possible values and their meanings.

**Return type** list

`info`

Meaningful for PDF only: A dictionary containing various information. All fields are (unicode) strings.

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

**Return type** dict

**flags**

Meaningful for PDF only: An integer whose low order bits contain flags for how the annotation should be presented. See section [Annotation Flags](#) for details.

**Return type** int

**lineEnds**

Meaningful for PDF only: A 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 [Annotation Line End Styles](#) and the [Adobe PDF Reference 1.7](#), table 8.27 on page 630.

**Return type** tuple

**vertices**

PDF only: A list containing a variable number of point (“vertices”) coordinates (each given by a pair of floats) for various types of annotations:

- `Line` – the starting and ending coordinates (2 float pairs).
- `[2, 'FreeText', 'FreeTextCallout']` – 2 or 3 float pairs designating the starting, the (optional) knee point, and the ending coordinates.
- `PolyLine / Polygon` – the coordinates of the edges connected by line pieces (n float pairs for n points).
- text markup annotations – 4 float pairs specifying the `QuadPoints` of the marked text span (see [Adobe PDF Reference 1.7](#), page 634).
- `Ink` – list of one to many sublists of vertex coordinates. Each such sublist represents a separate line in the drawing.

**Return type** list

**widget**

PDF only: A class containing all properties of a **form field** – including the following three attributes. `None` for other annotation types.

**Return type** [Widget](#)

**widget\_name**

PDF only: The field name for an annotation of type `ANNOT_WIDGET`, `None` otherwise. Equals [Widget.field\\_name](#).

**Return type** str

**widget\_value**

PDF only: The field content for an annotation of type `ANNOT_WIDGET`. Is `None` for non-PDFs, other annotation types, or if no value has been entered. For button types the value will be `True` or `False`. Push button states have no permanent reflection in the file and are therefore always reported as `False`. For text, list boxes and combo boxes, a string is returned for single values. If multiple selections have been made (may happen for list boxes and combo boxes), a list of strings is returned. For list boxes and combo boxes, the selectable values are contained in [widget\\_choices](#) below. Equals [Widget.field\\_value](#).

**Return type** bool, str or list

**widget\_choices**

PDF only: Contains a list of selectable values for list boxes and combo boxes (annotation type `ANNOT_WIDGET`), else `None`. Equals [Widget.choice\\_values](#).

**Return type** list

`widget_type`

PDF only: The field type for an annotation of type `ANNOT_WIDGET`, else `None`.

**Return type** tuple

**Returns** a tuple (int, str). E.g. for a text field (3, 'Text') is returned. For a complete list see [Annotation Types](#). The first item equals `Widget.field_type`, and the second is `Widget.field_type_string`.

`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 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 [] 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

`xref`

The PDF cross reference number. Zero if not a PDF.

**Return type** int

`border`

Meaningful for PDF only: A dictionary containing border characteristics. It will be `None` for non-PDFs and an empty dictionary if no border information exists. The following keys can occur:

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

**Return type** dict

### 4.15.1 Example

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

```
doc = fitz.open("circle-in.pdf")
page = doc[0]
annot = page.firstAnnot
annot.setBorder({"dashes": [3]})

# set stroke and fill color to some blue
annot.setColors({"stroke":(0, 0, 1), "fill":(0.75, 0.8, 0.95)})
info = annot.info
```

(continues on next page)

(continued from previous page)

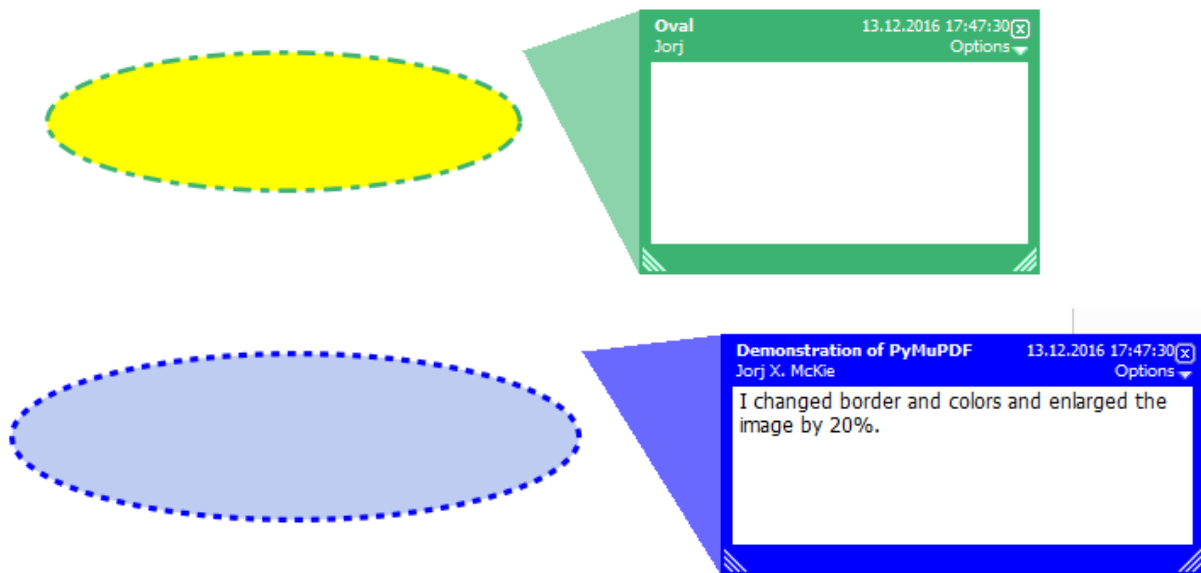
```

info["title"] = "Jorj X. McKie"          # set author

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



## 4.16 Widget

This class represents the properties of a PDF Form field, a “widget”. Fields are a special case of annotations, which allow users with limited permissions to enter information in a PDF. This is usually used for filling out forms.

### Class API

```
class Widget
```

`border_color`

A list of up to 4 floats defining the field’s border. Default value is `None` which causes border style and border width to be ignored.

`border_style`

A string defining the line style of the field’s border. See `Annot.border`. Default is “s” (“Solid”) –

a continuous line. Only the first character (upper or lower case) will be regarded when creating a widget.

`border_width`

A float defining the width of the border line. Default is 1.

`border_dashes`

A list of integers defining the dash properties of the border line. This is only meaningful if `border_style == "D"` and `border_color` is provided.

`choice_values`

A mandatory Python sequence of strings defining the valid choices of listboxes and comboboxes. Ignored for other field types. Equals `Annot.widget_choices`. The sequence must contain at least two items. When updating the widget, always the complete new list of values must be specified.

`field_name`

A mandatory string defining the field's name. Equals `Annot.widget_name`. No checking for duplicates takes place.

`field_value`

The value of the field. Equals `Annot.widget_value`.

`field_flags`

An integer defining a large amount of properties of a field. Handle this attribute with care

`field_type`

A mandatory integer defining the field type. This is a value in the range of 0 to 6. It cannot be changed when updating the widget.

`field_type_string`

A string describing (and derived from) the field type.

`fill_color`

A list of up to 4 floats defining the field's background color.

`button_caption`

For future use: the caption string of a button-type field.

`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 larger text amounts.

`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.

### 4.16.1 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 ( <b>default</b> )
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.

## 4.17 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.store_shrink()</code>	shrink the storables cache <sup>61</sup>
<code>Tools.fitz_stderr_reset()</code>	empty MuPDF messages on STDERR
<code>Tools.fitz_stdout_reset()</code>	empty MuPDF messages on STDOUT
<code>Tools.fitz_config</code>	configuration settings of PyMuPDF
<code>Tools.fitz_stderr</code>	sent to STDERR by MuPDF
<code>Tools.fitz_stdout</code>	sent to STDOUT by MuPDF
<code>Tools.store_maxsize</code>	maximum storables cache size
<code>Tools.store_size</code>	current storables cache size

---

<sup>61</sup> 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.



**Class API**

```
class Tools
```

```
    gen_id()
```

A convenience method returning a unique positive integer which will increase by 1 with 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:

- It is not part of MuPDF's global context and not threadsafe (because we do not support threads in PyMuPDF yet).
  - It is implemented as `int`. This means that the maximum number is  $2^{63} - 1$  (about  $9.223372e+18$ ) on most machines. Should this number ever be exceeded, the counter is reset to 1.
- 

**Return type** `int`

**Returns** a unique positive integer.

```
    store_shrink(percent)
```

Reduce the storables cache by a percentage of its current size.

**Parameters** `percent` (*int*) – the percentage of current size to free. If 100+ the store will be emptied, if zero, nothing will happen. MuPDF's caching strategy is “least recently used”, so low-usage elements get deleted first.

**Return type** `int`

**Returns** the new current store size. Depending on the situation, the size reduction may be larger than the requested percentage.

```
    fitz_stderr_reset()
```

Empty MuPDF messages on STDERR.

```
    fitz_stdout_reset()
```

Empty MuPDF messages on STDOUT.

```
    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
gprf	Ghostscript proofing 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 <sup>62</sup>
base14	Base-14 fonts (should always be true)

For an explanation of the term “TOFU” see [this Wikipedia article](#)<sup>60</sup>.

```
In [1]: import fitz
In [2]: TOOLS.fitz_config
Out[2]:
{'plotter-g': True,
 'plotter-rgb': True,
 'plotter-cmyk': True,
 'plotter-n': True,
 'pdf': True,
 'xps': True,
 'svg': True,
 'cbz': True,
 'img': True,
 'html': True,
 'epub': True,
 'gprf': False,
 'jpx': True,
 'js': True,
 'tofu': False,
 'tofu-cjk': True,
```

(continues on next page)

<sup>62</sup> 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.

<sup>60</sup> [https://en.wikipedia.org/wiki/Noto\\_fonts](https://en.wikipedia.org/wiki/Noto_fonts)

(continued from previous page)

```
'tofu-cjk-ext': False,
'tofu-cjk-lang': False,
'tofu-emoji': False,
'tofu-historic': False,
'tofu-symbol': False,
'tofu-sil': False,
'icc': False,
'py-memory': True, # (False if Python 2)
'base14': True}
```

**Return type** dict`fitz_stderr`

Contains all warnings and error messages issued by the underlying C-library MuPDF. Use it as a reference e.g. for diagnostics purposes. More often than not they can safely be ignored.

**Return type** unicode`fitz_stdout`

Contains MuPDF output sent to STDOUT.

**Return type** unicode`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

### 4.17.1 Example Session

```
>>> import fitz
# print the maximum and current cache sizes
>>> fitz.TOOLS.store_maxsize
268435456
>>> fitz.TOOLS.store_size
0
>>> doc = fitz.open("demo1.pdf")
# pixmap creation puts lots of object in cache (text, images, fonts),
# apart from the pixmap itself
>>> pix = doc[0].getPixmap(alpha=False)
>>> fitz.TOOLS.store_size
454519
# release (at least) 50% of the storage
>>> fitz.TOOLS.store_shrink(50)
13471
>>> fitz.TOOLS.store_size
```

(continues on next page)

(continued from previous page)

```
13471
# get a few unique numbers
>>> fitz.TOOLS.gen_id()
1
>>> fitz.TOOLS.gen_id()
2
>>> fitz.TOOLS.gen_id()
3
# close document and see how much cache is still in use
>>> doc.close()
>>> fitz.TOOLS.store_size
0
>>>
```

## 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.

### 5.1 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.  $+$ ,  $-$ ,  $*$ ,  $/$  are defined for all classes. They do what you would expect from them.
5. Rectangles have two additional binary operators:  $\&$  (intersection) and  $|$  (union).
6. Binary operators fully support in-place operations: if  $\circ$  denotes any binary operator, then  $a \circ= b$  is the same as  $a = a \circ b$ .
7. For binary operations, the **second** operand may always be a number sequence of the same size as the left one. We allude to this fact by e.g. saying “x-like object” if a number sequence of same length as x is allowed.

### 5.2 Unary Operations

	Result
<code>bool(O)</code>	is false exactly if all components of “O” are zero.
<code>abs(O)</code>	Euclidean norm (square root of the sum of component squares), if “O” is a <i>Point</i> or a <i>Matrix</i> . For rectangles, the area is returned.
<code>+O</code>	copy of “O”.
<code>-O</code>	copy of “O” with negated components.
<code>~m</code>	inverse of <i>Matrix</i> “m”.

## 5.3 Binary Operations

For the operators `+`, `-`, `*`, `/`, the **second operand** may be a number, which will be applied to each component. Otherwise:

	Result
<code>a+b</code> , <code>a-b</code>	component-wise execution, “b” must be “a”-like.
<code>a*m</code> , <code>a/m</code>	“a” can be any geometry object and “m” must be matrix-like. “a/m” is always treated as “a*~m”. If “a” is a <b>point</b> or a <b>rectangle</b> , then “a.transform(m)” is executed. If “a” is a matrix, then matrix concatenation takes place.
<code>a&amp;b</code>	<b>intersection rectangle</b> : “a” must be a rectangle and “b” rect-like. Delivers the <b>largest rectangle</b> contained in both operands.
<code>a b</code>	<b>union rectangle</b> : “a” must be a rectangle, and “b” may be point-like or rect-like. Delivers the <b>smallest rectangle</b> containing both operands.
<code>b in a</code>	if “b” is a number, then “b in tuple(a)” is returned. If “b” is point-like or rect-like, then “a” must be a rectangle, and “a.contains(b)” is returned.
<code>a==b</code>	True if bool(a-b) is False (“b” may be “a”-like).

## LOW LEVEL FUNCTIONS AND CLASSES

Contains a number of functions and classes for the experienced user. To be used for special needs or performance requirements.

### 6.1 Functions

The following are miscellaneous functions on a fairly low-level technical detail.

Some functions provide detail access to PDF structures. Others are stripped-down, high performance versions of functions providing more information.

Yet others are handy, general-purpose utilities.

Function	Short Description
<i>Document.FontInfos</i>	PDF only: information on inserted fonts
<i>Annot._cleanContents()</i>	PDF only: clean the annot's /Contents objects
<i>ConversionHeader()</i>	return header string for getText methods
<i>ConversionTrailer()</i>	return trailer string for getText methods
<i>Document._delXmlMetadata()</i>	PDF only: remove XML metadata
<i>Document._deleteObject()</i>	PDF only: delete an object
<i>Document._getGCTXerrmsg()</i>	retrieve C-level exception message
<i>Document._getNewXref()</i>	PDF only: create and return a new XREF entry
<i>Document._getOLRootNumber()</i>	PDF only: return / create xref of /Outline
<i>Document._getPageObjNumber()</i>	PDF only: return xref and generation number of a page
<i>Document._getPageXref()</i>	PDF only: same as _getPageObjNumber()
<i>Document._getXmlMetadataXref()</i>	PDF only: return XML metadata xref number
<i>Document._getXrefLength()</i>	PDF only: return length of xref table
<i>Document._getXrefStream()</i>	PDF only: return content of a stream object
<i>Document._getXrefString()</i>	PDF only: return object definition "source"
<i>Document._updateObject()</i>	PDF only: insert or update a PDF object
<i>Document._updateStream()</i>	PDF only: replace the stream of an object
<i>Document.extractFont()</i>	PDF only: extract embedded font
<i>Document.extractImage()</i>	PDF only: extract embedded image
<i>Document.getCharWidths()</i>	PDF only: return a list of glyph widths of a font
<i>getPDFnow()</i>	return the current timestamp in PDF format
<i>getPDFstr()</i>	return PDF-compatible string
<i>Page._cleanContents()</i>	PDF only: clean the page's /Contents objects
<i>Page._getContents()</i>	PDF only: return a list of content numbers
<i>Page._setContents()</i>	PDF only: set page's /Contents object to specified xref

Continued on next page

Table 1 – continued from previous page

Function	Short Description
<i>Page.getDisplayList()</i>	create the page's display list
<i>Page.insertFont()</i>	PDF only: store a new font in the document
<i>Page.getTextBlocks()</i>	extract text blocks as a Python list
<i>Page.getTextWords()</i>	extract text words as a Python list
<i>Page.run()</i>	run a page through a device
<i>PaperSize()</i>	return width, height for a known paper format
<i>PaperRect()</i>	return rectangle for a known paper format
<i>paperSizes</i>	dictionary of pre-defined paper formats

**PaperSize(s)**

Convenience function to return width and height of a known paper format code. These values are given in pixels for the standard resolution 72 pixels = 1 inch.

Currently defined formats include 'A0' through 'A10', 'B0' through 'B10', 'C0' through 'C10', 'Card-4x6', 'Card-5x7', 'Commercial', 'Executive', 'Invoice', 'Ledger', 'Legal', 'Legal-13', 'Letter', 'Monarch' and 'Tabloid-Extra', each in either portrait or landscape format.

A format name must be supplied as a string (case **in** sensitive), optionally suffixed with "-L" (landscape) or "-P" (portrait). No suffix defaults to portrait.

**Parameters** *s* (*str*) – any format name from above (upper or lower case), like "A4" or "letter-l".

**Return type** tuple

**Returns** (width, height) of the paper format. For an unknown format (-1, -1) is returned. 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)
>>>
```

---

**paperSizes**

A dictionary of pre-defines paper formats. Used as basis for `:meth:`PaperSize``.

---

**getPDFnow()**

Convenience function to return the current local timestamp in PDF compatible format, e.g. `D:20170501121525-04'00'` for local datetime May 1, 2017, 12:15:25 in a timezone 4 hours westward of the UTC meridian.



**Return type** str

**Returns** current local PDF timestamp.

`getPDFstr(obj, brackets = True)`

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

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

**Return type** str

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

`ConversionHeader(output = "text", filename = "UNKNOWN")`

Return the header string required to make a valid document out of page text outputs.

**Parameters**

- `output` (*str*) – type of document. Use the same as the output parameter of `getText()`.
- `filename` (*str*) – optional arbitrary name to use in output types “json” and “xml”.

**Return type** str

`ConversionTrailer(output)`

Return the trailer string required to make a valid document out of page text outputs. See [Page.getText\(\)](#) for an example.

**Parameters** `output` (*str*) – type of document. Use the same as the output parameter of `getText()`.

**Return type** str

`Document._deleteObject(xref)`

PDF only: Delete an object given by its cross reference number.

**Parameters** `xref` (*int*) – the cross reference number. Must be within the document's valid xref range.

**Caution:** Only use with extreme care: this may make the PDF unreadable.

`Document._delXmlMetadata()`

Delete an object containing XML-based metadata from the PDF. (Py-) MuPDF does not support XML-based metadata. Use this if you want to make sure that the conventional metadata dictionary will be used exclusively. Many thirdparty PDF programs insert their own metadata in XML format and thus may override what you store in the conventional dictionary. This method deletes any such reference, and the corresponding PDF object will be deleted during next garbage collection of the file.

---

`Document._getXmlMetadataXref()`

Return the XML-based metadata object id from the PDF if present – also refer to `Document._delXmlMetadata()`. You can use it to retrieve the content via `Document._getXrefStream()` and then work with it using some XML software.

---

`Document._getPageObjNumber(pno)`  
or

`Document._getPageXref(pno)`

Return the XREF and generation number for a given page.

**Parameters** `pno` (*int*) – Page number (zero-based).

**Return type** *list*

**Returns** XREF and generation number of page `pno` as a list [`xref`, `gen`].

---

`Page.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.getTextBlocks(images = False)`

Extract all blocks of the page's *TextPage* as a Python list. Provides basic positioning information but at a much higher speed than `TextPage.extractDICT()`. The block sequence is as specified in the document. All lines of a block are concatenated into one string, separated by `\n`.

**Parameters** `images` (*bool*) – also extract image blocks. Default is false. This serves as a means to get complete page layout information. Only image metadata, **not the binary image data** itself is extracted, see below (use the resp. `Page.getText()` versions for accessing full information detail).

**Return type** *list*

**Returns**

a list whose items have the following entries.

- `x0`, `y0`, `x1`, `y1`: 4 floats defining the bbox of the block.
  - `text`: concatenated text lines in the block (*str*). If this is an image block, a text like this is contained: `<image: DeviceRGB, width 511, height 379, bpc 8>` (original image properties).
  - `block_n`: 0-based block number (*int*).
  - `type`: block type (*int*), 0 = text, 1 = image.
-

`Page.getTextWords()`

Extract all words of the page's *TextPage* as a Python list. A “word” in this context is any character string surrounded by spaces. Provides positioning information for each word, similar to information contained in *TextPage.extractDICT()* or *TextPage.extractXML()*, but more directly and at a much higher speed. The word sequence is as specified in the document. The accompanying bbox coordinates can be used to re-arrange the final text output to your liking. Block and line numbers help keeping track of the original position.

**Return type** list

#### Returns

a list whose items are lists with the following entries:

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

`Page.insertFont(fontname = "Helvetica", fontfile = None, idx = 0, set_simple = False)`

Store a new font for the page and return its XREF. If the page already references this font, it is a no-operation and just the XREF is returned.

#### Parameters

- `fontname (str)` – The reference name of the font. If the name does not occur in *Page.getFontList()*, then this must be either the name of one of the *PDF Base 14 Fonts*, or `fontfile` must also be given. Following this method, font name prefixed with a slash “/” can be used to refer to the font in text insertions. If it appears in the list, the method ignores all other parameters and exits with the xref number.
- `fontfile (str)` – font file name. This file will be embedded in the PDF.
- `idx (int)` – index of the font in the given file. Has no meaning and is ignored if `fontfile` is not specified. Default is zero. An invalid index will cause an exception.

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

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

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

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

**Return type** `int`

**Returns**

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

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

---

`Page.getDisplayList()`

Run a page through a list device and return its display list.

**Return type** `DisplayList`

**Returns** the display list of the page.

---

`Page._getContents()`

Return a list of xref numbers of `/Contents` objects belonging to the page. The length of this list will always be at least one (otherwise the PDF is damaged).

**Return type** `list`

**Returns** a list of xref integers.

Each page has one or more associated contents objects (streams) which contain PDF some operator syntax describing what appears where on the page (like text or images, etc. See the *Adobe PDF Reference 1.7*, chapter “Operator Summary”, page 985). This function only enumerates the number(s) of such objects. To get the actual stream source, use function `Document._getXrefStream()` with one of the numbers in this list. Use `Document._updateStream()` to replace the content<sup>6364</sup>.

---

`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 into one as in the following snippet:

```
>>> c = b""
>>> xreflist = page._getContents()
>>> for xref in xreflist: c += doc._getXrefStream(xref)
>>> doc._updateStream(xreflist[0], c)
>>> page._setContents(xreflist[0])
>>> # doc.save(..., garbage = 1) will remove the unused objects
```

---

<sup>63</sup> If a page has multiple contents streams, they are treated as being one logical stream when the document is processed by reader software. A single operator cannot be split between stream boundaries, but a single **instruction** may well be. E.g. invoking the display of an image looks like this: `q a b c d e f cm /imageid Do Q`. Any single of these items (PDF notation: “lexical tokens”) is always contained in one stream, but `q a b c d e f cm` may be in one and `/imageid Do Q` in the next one.

<sup>64</sup> Note that `/Contents` objects (similar to `/Resources`) may be **shared** among pages. A change to a contents stream may therefore affect other pages, too. To avoid this: (1) use `Page._cleanContents()`, (2) read the `/Contents` object (there will now be only one left), (3) make your changes.

**Parameters** `xref (int)` – the cross reference number of a `/Contents` object. An exception is raised if outside the valid xref range or not a stream object.

`Page._cleanContents()`

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

**Return type** `int`

**Returns** 0 on success.

`Annot._cleanContents()`

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

**Return type** `int`

**Returns** 0 if successful (exception raised otherwise).

`Document.getCharWidths(xref = 0, limit = 256)`

Return a list of character glyphs and their widths for a font that is present in the document. A font must be specified by its PDF cross reference number `xref`. This function is called automatically from `Page.insertText()` and `Page.insertTextbox()`. So you should rarely need to do this yourself.

**Parameters**

- `xref (int)` – cross reference number of a font embedded in the PDF. To find a font xref, use e.g. `doc.getPageFontList(pno)` of page number `pno` and take the first entry of one of the returned list entries.
- `limit (int)` – limits the number of returned entries. The default of 256 is enforced for all fonts that only support 1-byte characters, so-called “simple fonts” (checked by this method). All [PDF Base 14 Fonts](#) are simple fonts.

**Return type** `list`

**Returns** a list of `limit` tuples. Each character `c` has an entry `(g, w)` in this list with an index of `ord(c)`. Entry `g` (integer) of the tuple is the glyph id of the character, and float `w` is its normalized width. The actual width for some font-size can be calculated as `w * fontsize`. For simple fonts, the `g` entry can always be safely ignored. In all other cases `g` is the basis for graphically representing `c`.

This function calculates the pixel width of a string called `text`:

```
def pixlen(text, widthlist, fontsize):
    try:
        return sum([widthlist[ord(c)] for c in text]) * fontsize
    except IndexError:
        m = max([ord(c) for c in text])
        raise ValueError("max. code point found: %i, increase limit" % m)
```

---

`Document._getXrefString(xref)`

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

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

**Return type** `string`

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

---

`Document._getGCTXerrmsg()`

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

**Return type** `str`

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

---

`Document._getNewXref()`

Increase the XREF by one entry and return that number. This can then be used to insert a new object.

**Return type** `int`

**Returns** the number of the new XREF entry.

---

`Document._updateObject(xref, obj_str, page = None)`

Associate the object identified by string `obj_str` with the XREF number `xref`, which must already exist. If `xref` pointed to an existing object, this will be replaced with the new object. If a page object is specified, links and other annotations of this page will be reloaded after the object has been updated.

**Parameters**

- `xref (int)` – XREF number.
- `obj_str (str)` – a string containing a valid PDF object definition.
- `page (Page)` – a page object. If provided, indicates, that annotations of this page should be refreshed (reloaded) to reflect changes incurred with links and / or annotations.

**Return type** `int`

**Returns** zero if successful, otherwise an exception will be raised.

---

`Document._getXrefLength()`

Return length of XREF table.

**Return type** `int`

**Returns** the number of entries in the XREF table.

---

---

`Document._getXrefStream(xref)`

Return the decompressed stream of the object referenced by `xref`. For non-stream objects `None` is returned.

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

**Return type** `bytes`

**Returns** the (decompressed) stream of the object.

---

`Document._updateStream(xref, stream, new = False)`

Replace the stream of an object identified by `xref`. If the object has no stream, an exception is raised unless `new = True` is used. The function automatically performs a compress operation (“deflate”).

**Parameters**

- `xref (int)` – XREF number.
- `stream (bytes/bytearray)` – the new content of the stream.
- `new (bool)` – whether to force accepting the stream, and thus turning `xref` into a stream object.

This method is intended to manipulate streams containing PDF operator syntax (see pp. 985 of the [Adobe PDF Reference 1.7](#)) as it is the case for e.g. page content streams.

If you update a contents stream, you should use save parameter `clean = True`. This ensures consistency between PDF operator source and the object structure.

Example: Let us assume that you no longer want a certain image appear on a page. This can be achieved by deleting<sup>64</sup> 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<sup>65</sup> show the image as being referenced by the page. This save option will clean up any such mismatches.

---

`Document._getOLRootNumber()`

Return XREF number of the `/Outlines` root object (this is **not** the first outline entry!). If this object does not exist, a new one will be created.

**Return type** `int`

**Returns** XREF number of the `/Outlines` root object.

`Document.extractImage(xref = 0)`

PDF Only: Extract data and meta information of an image stored in the document. The output can directly be used to be stored as an image file, as input for PIL, [Pixmap](#) creation, etc. This method avoids using pixmaps wherever possible to present the image in its original format (e.g. as JPEG).

**Parameters** `xref (int)` – cross reference number of an image object. If the object is not an image or other errors occur, an empty dictionary is returned and no exception is generated. Must however be in range of valid PDF cross reference numbers.

---

<sup>65</sup> Resources objects are inheritable. This means that many pages can share one. Keeping a page’s `/Resources` object in sync with changes of its `/Contents` therefore may require creating an own `/Resources` object for the page. This can best be achieved by using `clean` when saving, or by invoking [Page.\\_cleanContents\(\)](#).

**Return type** *dict*

### Returns

a dictionary with the following keys

- `ext` (*str*) image type (e.g. 'jpeg'), usable as image file extension
- `smask` (*int*) xref number of a stencil (/SMask) image or zero
- `width` (*int*) image width
- `height` (*int*) image height
- `colorspace` (*int*) the image's pixmap.n number (indicative only: depends on whether internal pixmaps had to be used)
- `xres` (*int*) resolution in x direction
- `yres` (*int*) resolution in y direction
- `image` (*bytes*) image data, usable as image file content

```
>>> d = doc.extractImage(25)
>>> d
{}
>>> d = doc.extractImage(1373)
>>> d
{'ext': 'png', 'smask': 2934, 'width': 5, 'height': 629, 'colorspace': 3, 'xres': 96,
 'yres': 96, '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 `pix.getPNGData()`. Main differences are that `extractImage` (1) does not only deliver PNG image formats, (2) is much faster with JPEG images, (3) usually results in much less disk storage for extracted images, (4) generates an empty *dict* for non-image xrefs (generates no exception). Look at the following example images within the same PDF.

- PNG image at xref 1268 – 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
```

- JPEG image at xref 1186 – `Document.extractImage()` is thousands of times faster and produces a **much smaller** output:

```
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())
```

(continues on next page)



(continued from previous page)

```

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, saves allocation of large buffer areas.

#### Return type tuple

#### Returns

a tuple (basename, ext, subtype, buffer), where ext is a 3-byte suggested file extension (*str*), basename is the font's name (*str*), subtype is the font's type (e.g. "Type1") and buffer is a bytes object containing the font file's content (or `b""`). For possible extension values and their meaning see [Font File Extensions](#). Return details on error:

- (`""`, `""`, `""`, `b""`) – invalid xref or xref is not a (valid) font object.
- (basename, "n/a", "Type1", `b""`) – basename is one of the [PDF Base 14 Fonts](#), which cannot be extracted.

Example:

```

>>> # store font as an external file
>>> name, ext, buffer = doc.extractFont(4711)
>>> # assuming buffer is not None:
>>> ofile = open(name + "." + ext, "wb")
>>> ofile.write(buffer)
>>> ofile.close()

```

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

`Document.FontInfos`

Contains following information for any font inserted via [Page.insertFont\(\)](#):

- `xref` (*int*) – XREF number of the /Type/Font object.
- `info` (*dict*) – detail font information with the following keys:
  - `name` (*str*) – name of the basefont
  - `idx` (*int*) – index number for multi-font files

- type (*str*) – font type (like “TrueType”, “Type0”, etc.)
- ext (*str*) – extension to be used, when font is extracted to a file (see [Font File Extensions](#)).
- glyphs (*list*) – list of glyph numbers and widths (filled by textinsertion methods).

**Return type** list

## 6.2 Device

The different format handlers (pdf, xps, etc.) interpret pages to a “device”. Devices are the basis for everything that can be done with a page: rendering, text extraction and searching. The device type is determined by the selected construction method.

### Class API

```
class Device
```

```
__init__(self, object, clip)
```

Constructor for either a pixel map or a display list device.

#### Parameters

- object (*Pixmap* or *DisplayList*) – either a Pixmap or a DisplayList.
- clip (*IRect*) – An optional *IRect* for Pixmap devices to restrict rendering to a certain area of the page. If the complete page is required, specify None. For display list devices, this parameter must be omitted.

```
__init__(self, textpage, flags = 0)
```

Constructor for a text page device.

#### Parameters

- textpage (*TextPage*) – TextPage object
- flags (*int*) – control the way how text is parsed into the text page. Currently 3 options can be coded into this parameter, see [Preserve Text Flags](#). To set these options use something like flags = 0 | TEXT\_PRESERVE\_LIGATURES | ....

---

**Note:** In higher level code (*Page.getText()*, *Document.getPageText()*), the following decisions for creating text devices have been implemented: (1) TEXT\_PRESERVE\_LIGATURES and TEXT\_PRESERVE\_WHITESPACES are always set, (2) TEXT\_PRESERVE\_IMAGES is set for JSON and HTML, otherwise off.

---

## 6.3 DisplayList

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

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

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

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

Method	Short Description
<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 – output of `page.bound()`.

**Return type** `DisplayList`

```
run(device, matrix, area)
```

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

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

### Parameters

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

```
getPixmap(matrix = fitz.Identity, colorspace = fitz.csRGB, alpha = 0, clip = None)
```

Run the display list through a draw device and return a pixmap.

### Parameters

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

**Return type** *Pixmap*

**Returns** pixmap of the display list.

```
getTextPage(flags)
```

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

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

**Return type** [TextPage](#)

**Returns** text page of the display list.

`rect`

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

**Type** [Rect](#)

## 6.4 TextPage

This class represents text and images shown on a document page. All MuPDF document types are supported.

Method	Short Description
<code>TextPage.extractText()</code>	Extract the page’s plain text
<code>TextPage.extractTEXT()</code>	synonym of previous
<code>TextPage.extractHTML()</code>	Extract the page’s content in HTML format
<code>TextPage.extractJSON()</code>	Extract the page’s content in JSON format
<code>TextPage.extractXHTML()</code>	Extract the page’s content in XHTML format
<code>TextPage.extractXML()</code>	Extract the page’s text in XML format
<code>TextPage.extractDICT()</code>	Extract the page’s content in <i>dict</i> format
<code>TextPage.extractRAWDICT()</code>	Extract the page’s content in <i>dict</i> format
<code>TextPage.search()</code>	Search for a string in the page

### Class API

```
class TextPage
```

```
    extractText()
```

```
    extractTEXT()
```

Extract all text from a `TextPage` object. Returns a string of the page’s complete text. The text is UTF-8 unicode and in the same sequence as specified at the time of document creation.

**Return type** `str`

```
    extractHTML()
```

Extract all text and images 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()
```

Extract content as a Python dictionary. Provides same information detail as HTML. See below for the structure.

**Return type** dict

`extractJSON()`

Extract content as a string in JSON format. Created by `json.dumps(TextPage.extractDICT())`. It is included only for backlevel compatibility. You will probably use this method ever only for outputting the result in some text file or the like.

**Return type** str

`extractXHTML()`

Extract all text 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()`

Extract all text in XML format. This contains complete formatting information about every single character on the page: font, size, line, paragraph, location, etc. Contains no images. You need an XML package to interpret the output in Python.

**Return type** str

`extractRAWDICT()`

Extract content as a Python dictionary – technically similar to `extractDICT()`, and it contains that information as a subset (including any images). It provides additional detail down to each character, which makes using XML obsolete in many cases. See below for the structure.

**Return type** dict

`search(string, hit_max = 16, quads = False)`

Search for `string` and return a list of found locations.

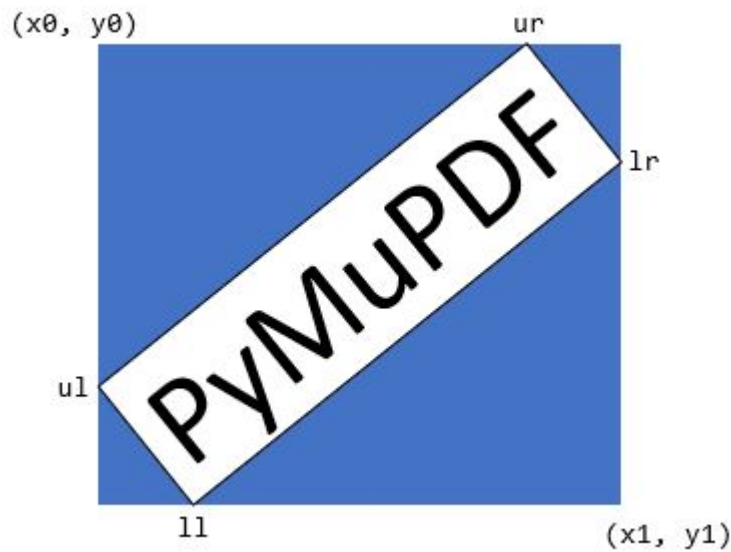
**Parameters**

- `string (str)` – the string to search for.
- `hit_max (int)` – maximum number of accepted hits (default 16).
- `quads (bool)` – return quadrilaterals instead of rectangles.

**Return type** list

**Returns** a list of `Rect` or `Quad` objects, each surrounding a found `string` occurrence.

Example: If the search for string “pymupdf” contains a hit like shown, then the corresponding entry will either be the blue rectangle, or, if `quads` was specified, `Quad(ul, ur, ll, lr)`.



---

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

---

### 6.4.1 Dictionary Structure of `extractDict()` and `extractRawDict()`

#### 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

#### Block Dictionaries

Blocks come in two different formats: **image blocks** and **text blocks**.

##### Image block:

Key	Value
type	1 = image ( <i>int</i> )
bbox	block / image rectangle, formatted as <code>list(fitz.Rect)</code>
ext	image type ( <i>str</i> ), as its file extension, see below
width	original image width ( <i>float</i> )
height	original image height ( <i>float</i> )
image	image content ( <i>bytes/bytearray</i> )

Possible values of key "ext" are "bmp", "gif", "jpeg", "jpx" (JPEG 2000), "jxr" (JPEG XR), "png", "pnm", and "tiff".

**Note:** All of the above values may be zero or contain empty objects respectively. In an effort to provide complete information we may return entries like `{'type': 1, 'bbox': [0.0, 0.0, 0.0, 0.0], 'width': 0, 'height': 0, 'ext': 'png', 'image': b''}`.

#### Text block:

Key	Value
type	0 = text ( <i>int</i> )
bbox	block rectangle, formatted as <code>list(fitz.Rect)</code>
lines	<i>list</i> of text line dictionaries

#### Line Dictionary

Key	Value
bbox	line rectangle, formatted as <code>list(fitz.Rect)</code>
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 beta is the writing angle relative to the horizontal.

#### Span Dictionary

Spans contain the actual text. In contrast to MuPDF versions prior to v1.12, a span no longer includes positioning information. Therefore, to reconstruct the text of a line, the text pieces of all spans must be concatenated. A span since v1.12 also contains font information. A line contains more than one span only, if the font or its attributes of the text are changing.

Key	Value
font	font name ( <i>str</i> )
size	font size ( <i>float</i> )
flags	font characteristics ( <i>int</i> )
text	(only for <code>extractDICT()</code> ) text ( <i>str</i> )
chars	(only for <code>extractRAWDICT()</code> ) <i>list</i> of character dictionaries

flags is an integer, encoding bools of font properties:

- bit 0: superscripted ( $2^0$ )
- bit 1: italic ( $2^1$ )
- bit 2: serified ( $2^2$ )
- bit 3: monospaced ( $2^3$ )
- bit 4: bold ( $2^4$ )

Test these characteristics like so:

```
>>> if flags & 2**0: print("super")
>>> if flags & 2**1: print("italic")
>>> if flags & 2**2: print("serif")
>>> # etc.
>>>
```

**Character Dictionary for `extractRAW_DICT()`**

Key	Value
bbox	character rectangle, formatted as <code>list(fitz.Rect)</code>
c	the character (unicode)
origin	<i>tuple</i> coordinates of the bottom left point

## 6.5 Working together: `DisplayList` and `TextPage`

Here are some instructions on how to use these classes together.

In some situations, performance improvements may be achievable, when you fall back to the detail level explained here.

### 6.5.1 Create a `DisplayList`

A *`DisplayList`* represents an interpreted document page. Methods for pixmap creation, text extraction and text search are – behind the curtain – all using the page's display list to perform their tasks. If a page must be rendered several times (e.g. because of changed zoom levels), or if text search and text extraction should both be performed, overhead can be saved, if the display list is created only once and then used for all other tasks.

```
>>> dl = page.getDisplayList()           # create the display list
```

You can also create display lists for many pages “on stack” (in a list), may be during document open, 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.

### 6.5.2 Generate Pixmap

The following creates a Pixmap from a *`DisplayList`*. Parameters are the same as for *`Page.getPixmap()`*.

```
>>> pix = dl.getPixmap()                 # create the page's pixmap
```

The execution time of this statement may be up to 50% shorter than that of *`Page.getPixmap()`*.



### 6.5.3 Perform Text Search

With the display list from above, we can also search for text.

For this we need to create a *TextPage*.

```
>>> tp = dl.getTextPage()           # display list from above
>>> rlist = tp.search("needle")     # look up "needle" locations
>>> for r in rlist:                 # work with the found locations, e.g.
    pix.invertIRect(r.irect)        # invert colors in the rectangles
```

### 6.5.4 Extract Text

With the same *TextPage* object from above, we can now immediately use any or all of the 5 text extraction methods.

---

**Note:** Above, we have created our text page without argument. This leads to a default argument of 3 = `fitz.TEXT_PRESERVE_LIGATURES | fitz.TEXT_PRESERVE_WHITESPACE`, IAW images will **not** be extracted – see below.

---

```
>>> txt = tp.extractText()          # plain text format
>>> json = tp.extractJSON()         # json format
>>> html = tp.extractHTML()         # HTML format
>>> xml = tp.extractXML()           # XML format
>>> xhtml = tp.extractXHTML()       # XHTML format
```

### 6.5.5 Further Performance improvements

#### Pixmap

As explained in the *Page* chapter:

If you do not need transparency set `alpha = 0` when creating pixmaps. This will save 25% memory (if RGB, the most common case) and possibly 5% execution time (depending on the GUI software).

#### TextPage

If you do not need images extracted alongside the text of a page, you can set the following option:

```
>>> flags = fitz.TEXT_PRESERVE_LIGATURES | fitz.TEXT_PRESERVE_WHITESPACE
>>> tp = dl.getTextPage(flags)
```

This will save ca. 25% overall execution time for the HTML, XHTML and JSON text extractions and **hugely** reduce the amount of storage (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
```



## CONSTANTS AND ENUMERATIONS

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

### 7.1 Constants

`Base14_Fonts`

Predefined Python list of valid *PDF Base 14 Fonts*.

**Return type** list

`csRGB`

Predefined RGB colorspace `fitz.Colorspace(fitz.CS_RGB)`.

**Return type** *Colorspace*

`csGRAY`

Predefined GRAY colorspace `fitz.Colorspace(fitz.CS_GRAY)`.

**Return type** *Colorspace*

`csCMYK`

Predefined CMYK colorspace `fitz.Colorspace(fitz.CS_CMYK)`.

**Return type** *Colorspace*

`CS_RGB`

1 – Type of *Colorspace* is RGBA

**Return type** int

`CS_GRAY`

2 – Type of *Colorspace* is GRAY

**Return type** int

`CS_CMYK`

3 – Type of *Colorspace* is CMYK

**Return type** int

`VersionBind`

'x.xx.x' – version of PyMuPDF (these bindings)

**Return type** string

`VersionFitz`

'x.xxx' – version of MuPDF

**Return type** string

VersionDate

ISO timestamp YYYY-MM-DD HH:MM:SS when these bindings were built.

**Return type** string

---

**Note:** The docstring of `fitz` contains information of the above which can be retrieved like so: `print(fitz.__doc__)`, and should look like: `PyMuPDF 1.10.0: Python bindings for the MuPDF 1.10 library, built on 2016-11-30 13:09:13.`

---

version

(VersionBind, VersionFitz, timestamp) – combined version information where `timestamp` is the generation point in time formatted as “YYYYMMDDhhmmss”.

**Return type** tuple

## 7.2 Font File Extensions

The table show file extensions you should use when extracting fonts from a PDF file.

Ext	Description
ttf	TrueType font
pfa	Postscript for ASCII font (various subtypes)
cff	Type1C font (compressed font equivalent to Type1)
cid	character identifier font (postscript format)
otf	OpenType font
n/a	one of the <a href="#">PDF Base 14 Fonts</a> (cannot be extracted)

## 7.3 Text Alignment

TEXT\_ALIGN\_LEFT

0 – align left.

TEXT\_ALIGN\_CENTER

1 – align center.

TEXT\_ALIGN\_RIGHT

2 – align right.

TEXT\_ALIGN\_JUSTIFY

3 – align justify.

## 7.4 Preserve Text Flags

Options controlling the amount of data a text device parses into a [TextPage](#).

TEXT\_PRESERVE\_LIGATURES

1 – If this option is activated ligatures are passed through to the application in their original form. If this option is deactivated ligatures are expanded into their constituent parts, e.g. the ligature `ffi` is expanded into three eparate characters `f`, `f` and `i`.

TEXT\_PRESERVE\_WHITESPACE

2 – If this option is activated whitespace is passed through to the application in its original form. If this option is deactivated any type of horizontal whitespace (including horizontal tabs) will be replaced with space characters of variable width.

TEXT\_PRESERVE\_IMAGES

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

## 7.5 Link Destination Kinds

Possible values of *linkDest.kind* (link destination kind). For details consult *Adobe PDF Reference 1.7*, chapter 8.2 on pp. 581.

LINK\_NONE

0 – No destination. Indicates a dummy link.

**Return type** int

LINK\_GOTO

1 – Points to a place in this document.

**Return type** int

LINK\_URI

2 – Points to a URI – typically a resource specified with internet syntax.

**Return type** int

LINK\_LAUNCH

3 – Launch (open) another file (of any “executable” type).

**Return type** int

LINK\_GOTOR

5 – Points to a place in another PDF document.

**Return type** int

## 7.6 Link Destination Flags

---

**Note:** The rightmost byte of this integer is a bit field, so test the truth of these bits with the `&` operator.

---

LINK\_FLAG\_L\_VALID

1 (bit 0) Top left x value is valid

**Return type** bool

LINK\_FLAG\_T\_VALID

2 (bit 1) Top left y value is valid

**Return type** bool

LINK\_FLAG\_R\_VALID

4 (bit 2) Bottom right x value is valid

**Return type** bool

LINK\_FLAG\_B\_VALID

8 (bit 3) Bottom right y value is valid

**Return type** bool

LINK\_FLAG\_FIT\_H

16 (bit 4) Horizontal fit

**Return type** bool

LINK\_FLAG\_FIT\_V

32 (bit 5) Vertical fit

**Return type** bool

LINK\_FLAG\_R\_IS\_ZOOM

64 (bit 6) Bottom right x is a zoom figure

**Return type** bool

## 7.7 Annotation Types

Possible values (integer) for PDF annotation types. See chapter 8.4.5, pp. 615 of the [Adobe PDF Reference 1.7](#) for more details.

ANNOT\_TEXT

0 – Text annotation

ANNOT\_LINK

1 – Link annotation

ANNOT\_FREETEXT

2 – Free text annotation

ANNOT\_LINE

3 – Line annotation

ANNOT\_SQUARE

4 – Square annotation

ANNOT\_CIRCLE

5 – Circle annotation

ANNOT\_POLYGON

6 – Polygon annotation

ANNOT\_POLYLINE

7 – PolyLine annotation

ANNOT\_HIGHLIGHT

8 – Highlight annotation

ANNOT\_UNDERLINE

9 – Underline annotation

ANNOT\_SQUIGGLY

10 – Squiggly-underline annotation

ANNOT\_STRIKEOUT

11 – Strikeout annotation

ANNOT\_STAMP  
12 – Rubber stamp annotation

ANNOT\_CARET  
13 – Caret annotation

ANNOT\_INK  
14 – Ink annotation

ANNOT\_POPUP  
15 – Pop-up annotation

ANNOT\_FILEATTACHMENT  
16 – File attachment annotation

ANNOT\_SOUND  
17 – Sound annotation

ANNOT\_MOVIE  
18 – Movie annotation

ANNOT\_WIDGET  
19 – Widget annotation. This annotation comes with the following subtypes:

ANNOT\_WG\_NOT\_WIDGET  
-1 not a widget

ANNOT\_WG\_PUSHBUTTON  
0 PushButton

ANNOT\_WG\_CHECKBOX  
1 CheckBox

ANNOT\_WG\_RADIOBUTTON  
2 RadioButton

ANNOT\_WG\_TEXT  
3 Text

ANNOT\_WG\_LISTBOX  
4 ListBox

ANNOT\_WG\_COMBOBOX  
5 ComboBox

ANNOT\_WG\_SIGNATURE  
6 Signature

ANNOT\_SCREEN  
20 – Screen annotation

ANNOT\_PRINTERMARK  
21 – Printers mark annotation

ANNOT\_TRAPNET  
22 – Trap network annotation

ANNOT\_WATERMARK  
23 – Watermark annotation

ANNOT\_3D  
24 – 3D annotation

## 7.8 Annotation Flags

Possible mask values for PDF annotation flags.

---

**Note:** Annotation flags is a bit field, so test the truth of its bits with the `&` operator. When changing flags for an annotation, use the `|` operator to combine several values. The following descriptions were extracted from the *Adobe PDF Reference 1.7*, pages 608 pp.

---

### ANNOT\_XF\_Invisible

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

### ANNOT\_XF\_Hidden

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

### ANNOT\_XF\_Print

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

### ANNOT\_XF\_NoZoom

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

### ANNOT\_XF\_NoRotate

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

### ANNOT\_XF\_NoView

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

### ANNOT\_XF\_ReadOnly

64 – If set, do not allow the annotation to interact with the user. The annotation may be displayed or printed (depending on the settings of the NoView and Print flags) but should not respond to mouse clicks or change its appearance in response to mouse motions.

### ANNOT\_XF\_Locked

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

### ANNOT\_XF\_ToggleNoView

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

### ANNOT\_XF\_LockedContents

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



## 7.9 Stamp Annotation Icons

MuPDF has defined the following icons for rubber stamp annotations.

STAMP_Approved	0 APPROVED
STAMP_AsIs	1 AS IS
STAMP_Confidential	2 CONFIDENTIAL
STAMP_Departmental	3 DEPARTMENTAL
STAMP_Experimental	4 EXPERIMENTAL
STAMP_Expired	5 EXPIRED
STAMP_Final	6 FINAL
STAMP_ForComment	7 FOR COMMENT
STAMP_ForPublicRelease	8 FOR PUBLIC RELEASE
STAMP_NotApproved	9 NOT APPROVED
STAMP_NotForPublicRelease	10 NOT FOR PUBLIC RELEASE
STAMP_Sold	11 SOLD
STAMP_TopSecret	12 TOP SECRET
STAMP_Draft	13 DRAFT

## 7.10 Annotation Line End Styles

The following descriptions are taken from the [Adobe PDF Reference 1.7](#) Table 8.27 on page 630. The visualizations are either dynamically created by PDF viewers, or explicitly hardcoded by the PDF generator software. Only 'FreeText', 'Line', 'PolyLine', and 'Polygon' annotation types can have these properties.

ANNOT_LE_None	0 – No line ending.
ANNOT_LE_Square	1 – A square filled with the annotation's interior color, if any.
ANNOT_LE_Circle	2 – A circle filled with the annotation's interior color, if any.

ANNOT\_LE\_Diamond

3 – A diamond shape filled with the annotation’s interior color, if any.

ANNOT\_LE\_OpenArrow

4 – Two short lines meeting in an acute angle to form an open arrowhead.

ANNOT\_LE\_ClosedArrow

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

ANNOT\_LE\_Butt

6 – (PDF 1.5) A short line at the endpoint perpendicular to the line itself.

ANNOT\_LE\_ROpenArrow

7 – (PDF 1.5) Two short lines in the reverse direction from OpenArrow.

ANNOT\_LE\_RClosedArrow

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

ANNOT\_LE\_Slash

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

## 7.11 PDF Form Field Flags

Bit positions in an integer (called `/Ff` in *Adobe PDF Reference 1.7*) controlling a wide range of PDF form field (“widget”) behaviours.

### 7.11.1 Common to all field types

WIDGET\_Ff\_ReadOnly

1 content cannot be changed

WIDGET\_Ff\_Required

2 must enter

WIDGET\_Ff\_NoExport

4 not available for export

### 7.11.2 Text fields

WIDGET\_Ff\_Multiline

4096 allow for line breaks

WIDGET\_Ff\_Password

8192 do not show entered text

WIDGET\_Ff\_FileSelect

1048576 file select field

WIDGET\_Ff\_DoNotSpellCheck

4194304 suppress spell checking

WIDGET\_Ff\_DoNotScroll

8388608 do not scroll screen automatically

WIDGET\_Ff\_Comb  
16777216

WIDGET\_Ff\_RichText  
33554432 richt text field

### 7.11.3 Button fields

WIDGET\_Ff\_NoToggleToOff  
16384 do not toggle off

WIDGET\_Ff\_Radio  
32768 make this a radio button (caution: overrides field type!)

WIDGET\_Ff\_Pushbutton  
65536 make this a push button (caution: overrides field type!)

WIDGET\_Ff\_RadioInUnison  
33554432 controls multiple radio buttons in a group (unsupported by PyMuPDF)

### 7.11.4 Choice fields

WIDGET\_Ff\_Combo  
131072 make this combo box (caution: overrides field type!)

WIDGET\_Ff\_Edit  
262144 make choice field editable (do not restrict values to value list)

WIDGET\_Ff\_Sort  
524288 sort value list for display

WIDGET\_Ff\_MultiSelect  
2097152 make multiple choice fields selectable

WIDGET\_Ff\_CommitOnSelCHange  
67108864 changing selected choice values counts as data entered



## COLOR DATABASE

Since the introduction of methods involving colors (like `Page.drawCircle()`), a requirement may be to have access to predefined colors.

The fabulous GUI package `wxPython`<sup>66</sup> 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.

### 8.1 Function `getColor()`

As the color database may not be needed very often, one additional import statement seems acceptable to get access to it:

```
>>> # "getColor" is the only method you really need
>>> from fitz.utils import getColor
>>> getColor("aliceblue")
(0.9411764705882353, 0.9725490196078431, 1.0)
>>> #
>>> # to get a list of all existing names
>>> from fitz.utils import getColorList
>>> cl = getColorList()
>>> cl
['ALICEBLUE', 'ANTIQUEWHITE', 'ANTIQUEWHITE1', 'ANTIQUEWHITE2', 'ANTIQUEWHITE3',
'ANTIQUEWHITE4', 'AQUAMARINE', 'AQUAMARINE1'] ...
>>> #
>>> # to see the full integer color coding
>>> from fitz.utils import getColorInfoList
>>> il = getColorInfoList()
>>> il
[('ALICEBLUE', 240, 248, 255), ('ANTIQUEWHITE', 250, 235, 215),
('ANTIQUEWHITE1', 255, 239, 219), ('ANTIQUEWHITE2', 238, 223, 204),
('ANTIQUEWHITE3', 205, 192, 176), ('ANTIQUEWHITE4', 139, 131, 120),
('AQUAMARINE', 127, 255, 212), ('AQUAMARINE1', 127, 255, 212)] ...
```

<sup>66</sup> <https://wxpython.org/>

## 8.2 Printing the Color Database

If you want to actually see how the many available colors look like, use scripts `colordbRGB.py`<sup>67</sup> or `colordbHSV.py`<sup>68</sup> 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.



<sup>67</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/colordbRGB.py>

<sup>68</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/colordbHSV.py>

## 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, <b>many</b> links / bookmarks
Evolution.pdf	13.497.490	75	15	118	1,1%	176	PDF 1.4	graphics oriented
PyMuPDF.pdf	479.011	47	60	491	13,2%	10	PDF 1.4	text oriented, many links
sdw_2015_01.pdf	14.668.972	100	36	0	2,5%	143	PDF 1.3	graphics oriented
sdw_2015_02.pdf	13.295.864	100	38	0	2,7%	130	PDF 1.4	graphics oriented
sdw_2015_03.pdf	21.224.417	108	35	0	1,9%	192	PDF 1.4	graphics oriented
sdw_2015_04.pdf	15.242.911	108	37	0	2,7%	138	PDF 1.3	graphics oriented
sdw_2015_05.pdf	16.495.887	108	43	0	2,4%	149	PDF 1.4	graphics oriented
sdw_2015_06.pdf	23.447.046	100	38	0	1,6%	229	PDF 1.4	graphics oriented
sdw_2015_07.pdf	14.106.982	100	38	2	2,6%	138	PDF 1.4	graphics oriented
sdw_2015_08.pdf	12.321.995	100	37	0	3,0%	120	PDF 1.4	graphics oriented
sdw_2015_09.pdf	23.409.625	100	37	0	1,5%	229	PDF 1.4	graphics oriented
sdw_2015_10.pdf	18.706.394	100	24	0	2,0%	183	PDF 1.5	graphics oriented
sdw_2015_11.pdf	25.624.266	100	20	0	1,5%	250	PDF 1.4	graphics oriented
sdw_2015_12.pdf	19.111.666	108	36	0	2,1%	173	PDF 1.4	graphics oriented

Decimal point and comma follow European convention

E.g. Adobe.pdf and PyMuPDF.pdf are clearly text oriented, all other files contain many more images.

### 9.1 Part 1: Parsing

How fast is a PDF file read and its content parsed for further processing? The sheer parsing performance cannot directly be compared, because batch utilities always execute a requested task completely, in one

go, front to end. `pdfrw` too, has a lazy strategy for parsing, meaning it only parses those parts of a document that are required in any moment.

To yet find an answer to the question, we therefore measure the time to copy a PDF file to an output file with each tool, and doing nothing else.

### These were the tools

All tools are either platform independent, or at least can run both, on Windows and Unix / Linux (`pdftk`).

**Poppler** is missing here, because it specifically is a Linux tool set, although we know there exist Windows ports (created with considerable effort apparently). Technically, it is a C/C++ library, for which a Python binding exists – in so far somewhat comparable to PyMuPDF. But Poppler in contrast is tightly coupled to **Qt** and **Cairo**. We may still include it in future, when a more handy Windows installation is available. We have seen however some [analysis](#)<sup>69</sup>, that hints at a much lower performance than MuPDF. Our comparison of text extraction speeds also show a much lower performance of Poppler's PDF code base **Xpdf**.

Image rendering of MuPDF also is about three times faster than the one of Xpdf when comparing the command line tools `mudraw` of MuPDF and `pdftopng` of Xpdf – see part 3 of this chapter.

Tool	Description
PyMuPDF	tool of this manual, appearing as “fitz” in reports
pdfrw	a pure Python tool, is being used by <code>rst2pdf</code> , has interface to ReportLab
PyPDF2	a pure Python tool with a very complete function set
pdftk	a command line utility with numerous functions

This is how each of the tools was used:

#### PyMuPDF:

```
doc = fitz.open("input.pdf")
doc.save("output.pdf")
```

#### pdfrw:

```
doc = PdfReader("input.pdf")
writer = PdfWriter()
writer.trailer = doc
writer.write("output.pdf")
```

#### PyPDF2:

```
pdfmerge = PyPDF2.PdfFileMerger()
pdfmerge.append("input.pdf")
pdfmerge.write("output.pdf")
pdfmerge.close()
```

#### pdftk:

```
pdftk input.pdf output output.pdf
```

### Observations

These are our run time findings (in **seconds**, please note the European number convention: meaning of decimal point and comma is reversed):

---

<sup>69</sup> <http://hzqtc.github.io/2012/04/poppler-vs-mupdf.html>



Runtimes	Tool			
File	fitz	pdfrw	pdftk	PyPDF2
Adobe.pdf	4,96	20,72	136,34	683,27
Evolution.pdf	0,40	0,41	1,22	0,94
PyMuPDF.pdf	0,04	0,19	1,03	0,97
sdw_2015_01.pdf	0,19	1,19	6,13	6,49
sdw_2015_02.pdf	0,23	1,52	7,74	7,02
sdw_2015_03.pdf	0,39	2,76	13,39	12,67
sdw_2015_04.pdf	0,25	2,14	8,55	7,50
sdw_2015_05.pdf	0,29	1,71	8,92	7,99
sdw_2015_06.pdf	0,53	3,30	16,05	15,56
sdw_2015_07.pdf	0,33	2,17	10,65	10,81
sdw_2015_08.pdf	0,29	2,01	9,65	9,39
sdw_2015_09.pdf	0,36	2,49	11,48	10,97
sdw_2015_10.pdf	0,27	1,87	3,31	6,74
sdw_2015_11.pdf	1,47	12,79	40,18	62,44
sdw_2015_12.pdf	0,39	2,21	10,40	10,19
<b>Total Times</b>	<b>10,40</b>	<b>57,46</b>	<b>285,04</b>	<b>852,96</b>

Time Ratios			
1,00	5,52	27,40	81,98
	1,00	4,96	14,84
		1,00	2,99
			1,00

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

Runtimes	Tool			
File	fitz	pdfrw	pdftk	PyPDF2
Evolution.pdf	0,40	0,41	1,22	0,94
PyMuPDF.pdf	0,04	0,19	1,03	0,97
sdw_2015_01.pdf	0,19	1,19	6,13	6,49
sdw_2015_02.pdf	0,23	1,52	7,74	7,02
sdw_2015_03.pdf	0,39	2,76	13,39	12,67
sdw_2015_04.pdf	0,25	2,14	8,55	7,50
sdw_2015_05.pdf	0,29	1,71	8,92	7,99
sdw_2015_06.pdf	0,53	3,30	16,05	15,56
sdw_2015_07.pdf	0,33	2,17	10,65	10,81
sdw_2015_08.pdf	0,29	2,01	9,65	9,39
sdw_2015_09.pdf	0,36	2,49	11,48	10,97
sdw_2015_10.pdf	0,27	1,87	3,31	6,74
sdw_2015_11.pdf	1,47	12,79	40,18	62,44
sdw_2015_12.pdf	0,39	2,21	10,40	10,19
<b>Gesamtergebnis</b>	<b>5,44</b>	<b>36,75</b>	<b>148,70</b>	<b>169,69</b>

Time Ratios			
1,00	6,75	27,32	31,18
	1,00	4,05	4,62
		1,00	1,14
			1,00

PyMuPDF is by far the fastest: on average 4.5 times faster than the second best (the pure Python tool `pdfrw`, **chapeau pdfrw!**), and almost 20 times faster than the command line tool `pdftk`.

Where PyMuPDF only requires less than 13 seconds to process all files, `pdftk` affords itself almost 4 minutes.

By far the slowest tool is PyPDF2 – it is more than 66 times slower than PyMuPDF and 15 times slower than `pdfrw`! The main reason for PyPDF2's bad look comes from the Adobe manual. It obviously is slowed down by the linear file structure and the immense amount of bookmarks of this file. If we take out this special case, then PyPDF2 is only 21.5 times slower than PyMuPDF, 4.5 times slower than `pdfrw` and 1.2 times slower than `pdftk`.

If we look at the output PDFs, there is one surprise:

Each tool created a PDF of similar size as the original. Apart from the Adobe case, PyMuPDF always created the smallest output.

Adobe's manual is an exception: The pure Python tools `pdfrw` and PyPDF2 **reduced** its size by more than 20% (and yielded a document which is no longer linearized)!

PyMuPDF and `pdftk` in contrast **drastically increased** the size by 40% to about 50 MB (also no longer linearized).

So far, we have no explanation of what is happening here.

## 9.2 Part 2: Text Extraction

We also have compared text extraction speed with other tools.

The following table shows a run time comparison. PyMuPDF's methods appear as “fitz (TEXT)” and “fitz (JSON)” respectively. The tool `pdftotext.exe` of the [Xpdf](#)<sup>70</sup> 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](#)<sup>71</sup>)
- **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
<b>Gesamtergebnis</b>	<b>21,92</b>	<b>23,08</b>	<b>24,82</b>	<b>57,10</b>	<b>1321,51</b>

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](#)

<sup>70</sup> <http://www.foolabs.com/xpdf/>

<sup>71</sup> <http://poppler.freedesktop.org/>

site<sup>72</sup>.

## 9.3 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):

---

<sup>72</sup> <http://www.unixuser.org/~euske/python/pdfminer/>

Render Speed	tool		
file	mudraw	pymupdf	xpdf
Adobe.pdf	105,09	110,66	505,27
Evolution.pdf	40,70	42,17	108,33
PyMuPDF.pdf	5,09	4,96	21,82
sdw_2015_01.pdf	29,77	30,40	76,81
sdw_2015_02.pdf	29,67	30,00	74,68
sdw_2015_03.pdf	32,67	32,88	85,89
sdw_2015_04.pdf	30,07	29,59	78,09
sdw_2015_05.pdf	31,37	31,39	77,56
sdw_2015_06.pdf	31,76	31,49	87,89
sdw_2015_07.pdf	33,33	34,58	78,74
sdw_2015_08.pdf	31,83	32,73	75,95
sdw_2015_09.pdf	36,92	36,77	84,37
sdw_2015_10.pdf	30,08	30,48	77,13
sdw_2015_11.pdf	33,21	34,11	80,96
sdw_2015_12.pdf	31,77	32,69	80,68
<b>Gesamtergebnis</b>	<b>533,33</b>	<b>544,90</b>	<b>1594,18</b>

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)?

### 10.1 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 font information and characters that share a common baseline.

### 10.2 Plain Text

This function extracts a page's plain **text in original order** as specified by the creator of the document (which may not equal a natural reading order).

An example output:

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

## 10.3 HTML

HTML output fully reflects the structure of the page’s TextPage – much like DICT or 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:

```
<div style="width:595pt;height:841pt">

<p style="top:189pt;left:195pt;"><b><span style="font-family:SFSX2488,serif;font-size:24.7871pt;">
↳PyMuPDF Documentation</span></b></p>
<p style="top:223pt;left:404pt;"><b><i><span style="font-family:SFS01728,serif;font-size:17.2154pt;">
↳"Release 1.12.0</span></i></b></p>
<p style="top:371pt;left:400pt;"><b><span style="font-family:SFSX1728,serif;font-size:17.2154pt;">
↳Jorj X. McKie</span></b></p>
<p style="top:637pt;left:448pt;"><b><span style="font-family:SFSX1200,serif;font-size:11.9552pt;">
↳Dec 04, 2017</span></b></p>
</div>
```

## 10.4 Controlling Quality of HTML Output

Though HTML output has improved a lot in MuPDF v1.12.0, it currently is not yet bug-free: we have found problems in the areas **font support** and **image positioning**.

- HTML text contains references to the fonts used of the original document. If these are not known to the browser (a fat chance!), it will replace them with his assumptions, which probably will let the result look awkward. This issue varies greatly by browser – on my Windows machine, MS Edge worked just fine, whereas Firefox looked horrible.
- For PDFs with a complex structure, images may not be positioned and / or sized correctly. This seems to be the case for rotated pages and pages, where the various possible page bbox variants do not coincide (e.g. MediaBox != CropBox). We do not know yet, how to address this – we filed a bug at MuPDF’s site.

To address the font issue, you can use a simple utility script to scan through the HTML file and replace font references. Here is a little example that replaces all fonts with one of the *PDF Base 14 Fonts*: serifed fonts will become “Times”, non-serifed “Helvetica” and monospaced will become “Courier”. Their respective variations for “bold”, “italic”, etc. are hopefully done correctly by your browser:

```
import sys
filename = sys.argv[1]
otext = open(filename).read()
pos1 = 0
font_serif = "font-family:Times"
font_sans = "font-family:Helvetica"
font_mono = "font-family:Courier"
found_one = False
# original html text string
# search start poition
# enter ...
# ... your choices ...
# ... here
# true if search successfull
```

(continues on next page)



(continued from previous page)

```

while True:
    pos0 = otext.find("font-family:", pos1)  # start of a font spec
    if pos0 < 0:                               # none found - we are done
        break
    pos1 = otext.find(";", pos0)              # end of font spec
    test = otext[pos0 : pos1]                 # complete font spec string
    testn = ""                               # the new font spec string
    if test.endswith(",serif"):               # font with serifs?
        testn = font_serif                   # use Times instead
    elif test.endswith(",sans-serif"):        # sans serifs font?
        testn = font_sans                    # use Helvetica
    elif test.endswith(",monospace"):         # monospaced font?
        testn = font_mono                   # becomes Courier

    if testn != "":                          # any of the above found?
        otext = otext.replace(test, testn)   # change the source
        found_one = True
        pos1 = 0                            # start over

if found_one:
    ofile = open(filename + ".html", "w")
    ofile.write(otext)
    ofile.close()
else:
    print("Warning: could not find any font specs!")

```

## 10.5 DICT (or JSON)

DICT (JSON) output fully reflects the structure of a TextPage and provides image content and position details (bbox – boundary boxes in pixel units) for every block and line. This information can be used to present text in another reading order if required (e.g. from top-left to bottom-right). Have a look at [PDF2textJS.py](#)<sup>73</sup>. Images are stored as bytes (bytearray in Python 2) for DICT output and base64 encoded strings for JSON output. Here is how this looks like:

```

In [2]: doc = fitz.open("pymupdf.pdf")
In [3]: page = doc[0]
In [4]: d = page.getText("dict")
In [5]: d
Out[5]:
{'width': 612.0,
'height': 792.0,
'blocks': [{ 'type': 1,
  'bbox': [344.25, 88.93597412109375, 540.0, 175.18597412109375],
  'width': 261,
  'height': 115,
  'ext': 'jpeg',
  'image': b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01 ... <more data> ...'},
{ 'type': 0,
  'lines': [{ 'wmode': 0,
    'dir': (1.0, 0.0),
    'spans': [{ 'font': 'ClearSans-Bold',

```

(continues on next page)

<sup>73</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/PDF2textJS.py>

(continued from previous page)

```

        'size': 24.787099838256836,
        'flags': 20,
        'text': 'PyMuPDF Documentation']],
    'bbox': (251.24600219726562,
            184.3526153564453,
            539.9661254882812,
            218.6648406982422)}],
    'bbox': (251.24600219726562,
            184.3526153564453,
            539.9661254882812,
            218.6648406982422)},
    {'type': 0,
     'lines': [{ 'wmode': 0,
                  'dir': (1.0, 0.0),
                  'spans': [{ 'font': 'ClearSans-BoldItalic',
                              'size': 17.21540069580078,
                              'flags': 22,
                              'text': 'Release 1.13.18'}]],
                  'bbox': (412.5299987792969,
                          220.4202880859375,
                          540.0100708007812,
                          244.234375)}],
     'bbox': (412.5299987792969,
            220.4202880859375,
            540.0100708007812,
            244.234375)},
    {'type': 0,
     'lines': [{ 'wmode': 0,
                  'dir': (1.0, 0.0),
                  'spans': [{ 'font': 'ClearSans-Bold',
                              'size': 17.21540069580078,
                              'flags': 20,
                              'text': 'Jorj X. McKie'}]],
                  'bbox': (432.9129943847656,
                          355.5234680175781,
                          534.0018310546875,
                          379.3543701171875)}],
     'bbox': (432.9129943847656,
            355.5234680175781,
            534.0018310546875,
            379.3543701171875)},
    {'type': 0,
     'lines': [{ 'wmode': 0,
                  'dir': (1.0, 0.0),
                  'spans': [{ 'font': 'ClearSans-Bold',
                              'size': 11.9552001953125,
                              'flags': 20,
                              'text': 'Aug 23, 2018'}]],
                  'bbox': (465.7779846191406,
                          597.5914916992188,
                          539.995849609375,
                          614.1408081054688)}],
     'bbox': (465.7779846191406,
            597.5914916992188,
            539.995849609375,
            614.1408081054688)}}

```

In [6]:

## 10.6 RAWDICT

This dictionary 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': 'PyMuPDF Documentation' above:

```
'chars': [{ 'c': 'P',
  'origin': (251.24600219726562, 211.052001953125),
  'bbox': (251.24600219726562,
    184.3526153564453,
    266.2421875,
    218.6648406982422)},
{ 'c': 'y',
  'origin': (266.2421875, 211.052001953125),
  'bbox': (266.2421875,
    184.3526153564453,
    279.3793640136719,
    218.6648406982422)},
{ 'c': 'M',
  'origin': (279.3793640136719, 211.052001953125),
  'bbox': (279.3793640136719,
    184.3526153564453,
    299.5560607910156,
    218.6648406982422)},
... <more character dicts> ...
{ 'c': 'o',
  'origin': (510.84130859375, 211.052001953125),
  'bbox': (510.84130859375,
    184.3526153564453,
    525.2426147460938,
    218.6648406982422)},
{ 'c': 'n',
  'origin': (525.2426147460938, 211.052001953125),
  'bbox': (525.2426147460938,
    184.3526153564453,
    539.9661254882812,
    218.6648406982422)}}}]
```

## 10.7 XML

The XML version extracts text (no images) with the detail level of RAWDICT:

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

(continues on next page)

(continued from previous page)

```

</line>
</block>
<block bbox="404.002 223.5048 523.30477 244.49039">
<line bbox="404.002 223.5048 523.30477 244.49039" wmode="0" dir="1 0">
<font name="SFS01728" size="17.2154">
<char bbox="404.002 223.5048 416.91358 244.49039" x="404.002" y="238.94702" c="R"/>
(... omitted data ...)
<char bbox="513.33706 223.5048 523.30477 244.49039" x="513.33706" y="238.94702" c="0"/>
</font>
</line>
</block>
(... omitted data ...)
</page>

```

**Note:** We have successfully tested `lxml`<sup>74</sup> to interpret this output.

## 10.8 XHTML

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

```

<div>
<p></p>
<p><b>PyMuPDF Documentation</b></p>
<p><b><i>Release 1.12.0</i></b></p>
<p><b>Jorj X. McKie</b></p>
<p><b>Dec 13, 2017</b></p>
</div>

```

## 10.9 Further Remarks

1. We have modified MuPDF’s **plain text** extraction: The original prints out every line followed by a newline character. This leads to a rather ragged, space-wasting look. We have combined all lines of a text block into one, separating lines by space characters. We also do not add extra newline characters at the end of blocks.
2. The extraction methods each have its own default behavior concerning images: “TEXT” and “XML” do not extract images, while the others do. On occasion it may make sense to **switch off images** for them, too. See chapter *Working together: DisplayList and TextPage* on how to achieve this. To **exclude images**, use an argument of 3 when you create the *TextPage*.
3. Apart from the above “standard” ones, we offer additional extraction methods *Page.getTextBlocks()* and *Page.getTextWords()* for performance reasons. They return lists of a page’s text blocks, resp. words. Each list item contains text accompanied by its rectangle (“bbox”, location on the page). This should help to resolve extraction issues around multi-column or boxed text.
4. For uttermost detail, down to the level of one character, use RAWDICT extraction.

<sup>74</sup> <https://pypi.org/project/lxml/>

## 10.10 Performance

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

To begin with, all methods are **very fast** in relation to other products out there in the market. In terms of processing speed, we couldn't find a faster (free) tool. Even the most detailed method, RAWDICT, processes all 1'310 pages of the [Adobe PDF Reference 1.7](#) in less than 9 seconds (simple text needs less than 2 seconds here).

Relative to each other, **“RAWDICT”** is about 4.6 times slower than **“TEXT”**, the others range between them. The following table shows **relative runtimes** with **“TEXT”** set to 1, measured across ca. 1550 text-heavy and 250 image-heavy pages.

Method	Time	Comments
TEXT	1.00	no images, plain text, line breaks
WORDS	1.07	no images, word level text with bboxes
BLOCKS	1.10	image bboxes (only), block level text with bboxes
XML	2.30	no images, char level text, layout and font details
DICT	2.68	<b>binary</b> images, span level text, layout and font details
XHTML	3.51	<b>base64</b> images, span level text, no layout info
HTML	3.60	<b>base64</b> images, span level text, layout and font details
RAWDICT	4.61	<b>binary</b> images, char level text, layout and font details

In versions prior to v1.13.1, JSON was a standalone extraction method. Since we have added the DICT extraction, JSON output is now created from it, using the **json** module contained in Python for serialization. We believe, DICT output is more handy for the programmer's purpose, because all of its information is directly usable – including images. Previously, for JSON, you had to base64-decode images before you could use them. We also have replaced the old “imgtype” dictionary key (an integer bit code) with the key “ext”, which contains the appropriate extension string for the image.

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



## APPENDIX 3: CONSIDERATIONS ON EMBEDDED FILES

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

### 11.1 General

Starting with version 1.4, PDF supports embedding arbitrary files as part (“Embedded File Streams”) of a PDF document file (see chapter 3.10.3, pp. 184 of the [Adobe PDF Reference 1.7](#)).

In many aspects, this is comparable to concepts also found in ZIP files or the OLE technique in MS Windows. PDF embedded files do, however, *not* support directory structures as does the ZIP format. An embedded file can in turn contain embedded files itself.

Advantages of this concept are that embedded files are under the PDF umbrella, benefitting from its permissions / password protection and integrity aspects: all files a PDF may reference or even be dependent on can be bundled into it and so form a single, consistent unit of information.

In addition to embedded files, PDF 1.7 adds *collections* to its support range. This is an advanced way of storing and presenting meta information (i.e. arbitrary and extensible properties) of embedded files.

### 11.2 MuPDF Support

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

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

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

### 11.3 PyMuPDF Support

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

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

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



## APPENDIX 4: ASSORTED TECHNICAL INFORMATION

### 12.1 PDF Base 14 Fonts

The following 14 builtin font names must be supported by every PDF viewer application. They are available as follows:

```
In [1]: import fitz
In [2]: fitz.Base14_fontnames
Out[2]:
('Courier',
 'Courier-Oblique',
 'Courier-Bold',
 'Courier-BoldOblique',
 'Helvetica',
 'Helvetica-Oblique',
 'Helvetica-Bold',
 'Helvetica-BoldOblique',
 'Times-Roman',
 'Times-Italic',
 'Times-Bold',
 'Times-BoldItalic',
 'Symbol',
 'ZapfDingbats')
```

---

### 12.2 Adobe PDF Reference 1.7

This PDF Reference manual published by Adobe is frequently quoted throughout this documentation. It can be viewed and downloaded from [here](http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf)<sup>75</sup>.

---

### 12.3 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.

---

<sup>75</sup> [http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf\\_reference\\_1-7.pdf](http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf)

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.

---

## 12.4 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.

---

## 12.5 Design of Method `Page.showPDFpage()`

### 12.5.1 Purpose and Capabilities

The method displays an image of a (“source”) page of another PDF document within a specified rectangle of the current (“containing”) 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 width-height-ratio (default) or not.
- Rectangle parameter `clip` controls which part of the source page to show, and hence can be used for cropping. Default is the full page.
- 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.

## 12.5.2 Technical Implementation

This is done using PDF “**Form XObjects**”, see section 4.9 on page 355 of *Adobe PDF Reference 1.7*. On execution of a `Page.showPDFpage(rect, src, pno, ...)`, the following things happen:

1. The `/Resources` and `/Contents` objects of page `pno` in document `src` are copied over to the current document, jointly creating a new **Form XObject** with the following properties. The PDF `xref` number of this object is returned by the method.
  - a. `/BBox` equals `/Mediabox` of the source page
  - b. `/Matrix` equals the identity matrix `[1 0 0 1 0 0]`
  - c. `/Resources` equals that of the source page. This involves a “deep-copy” of hierarchically nested other objects (including fonts, images, etc.). The complexity involved here is covered by MuPDF’s grafting<sup>76</sup> technique functions.
  - d. This is a stream object type, and its stream is exactly equal to the `/Contents` object of the source (if the source has multiple such objects, these are first concatenated and stored as one new stream into the new form XObject).
2. A second **Form XObject** is then created which the containing page uses to invoke the previous one. This object has the following properties:
  - a. `/BBox` equals the `/CropBox` of the source page (or `clip`, if specified).
  - b. `/Matrix` represents the mapping of `/BBox` to the display rectangle of the containing page (parameter 1 of `showPDFpage`).
  - 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 invoking page are now modified as follows.
  - a. Add an entry to the `/XObject` dictionary of `/Resources` with the name `fzFrm<n>` with an appropriately chosen integer `n` that makes this entry unique on the page.
  - b. Depending on `overlay`, prepend or append a new object to the page’s `/Contents` containing the statement `q /fzFrm<n> Do Q`.
4. Return `xref` to the caller.

Observe the following guideline for optimum results:

The second XObject is small (just about 270 bytes), specific to the containing rectangle, and therefore different each time.

If no precautions are taken, process **step 1** leads to another XObject on every invocation – even for the same source page. Its size may be several dozens of kilobytes large. To avoid identical source page copies, use parameter `reuse_xref = xref` with the `xref` value returned by previous executions. If `reuse_xref > 0`, the method will not create XObject 1 again, but instead just point to it via XObject 2. This significantly saves processing time, memory and disk usage.

If you forget to use `reuse_xref`, garbage collection (`mutool clean -ggg` or save option `garbage = 3`) can still take care of any duplicates.

<sup>76</sup> MuPDF supports “deep-copying” objects between PDF documents. To avoid duplicate data in the target, it uses so-called “graftmaps”, 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, as our tests have shown, because it prevents multiple copies of typically large and frequently referenced data, like images and fonts. Whether the target document **originally** had identical data is, however, not checked by this technique. Therefore, using save-option `garbage = 4` is still reasonable when copying to a non-empty target.

## 12.6 Redirecting Error and Warning Messages

In the past, MuPDF error and warning messages unavoidably were sent to the Operating System's files `STDOUT` or `STDERR`. Especially for interactive Python sessions, this was annoying, because important diagnostic information could remain unseen.

Another issue – frequently admonished by our users – was the occasionally large amount of warning messages spilled out – partly obscure to the developer, without apparent corrective action being possible or even required. Some examples are "warning: freetype getting character advance: invalid glyph index", or "warning: push viewport: 0 0 181 115" – the only possible comment to these was "so what?".

Since v1.14.0 we are capturing (hopefully) all warning and many error messages and store them away internally. A differentiation between warnings and errors is not possible, because MuPDF outputs both categories to `stderr`.

You can always empty or check this store of messages. It is kept as a unicode string which can be saved or printed. Look at chapter [Collection of Recipes](#) for an example.

## 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](#)<sup>77</sup> pages, but some duplication may still occur.

---

### 13.1 Images

---

#### 13.1.1 How to Make Images from Document Pages

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

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

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

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

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

---

#### 13.1.2 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.

---

<sup>77</sup> <https://github.com/rk700/PyMuPDF/wiki>

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

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

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

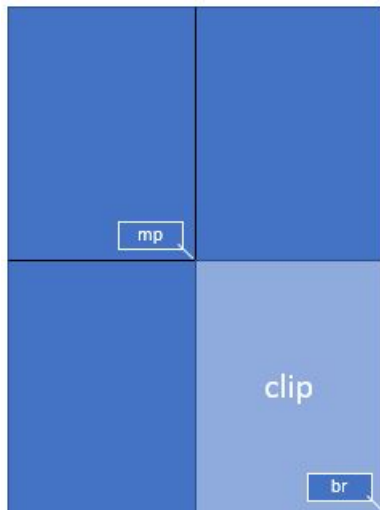
The resulting pixmap will be 4 times bigger than normal.

---

### 13.1.3 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.



```
>>> mat = fitz.Matrix(2, 2)                # zoom factor 2 in each direction
>>> rect = page.rect                        # page rectangle
>>> mp = rect.tl + (rect.br - rect.tl) * 0.5 # center of rect
>>> clip = fitz.Rect(mp, rect.br)           # clipping 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`.

---

### 13.1.4 How to Suppress Annotation Images

Normally, the pixmap of a page also includes the images of any annotations. There currently is no direct way to suppress this.



But it can be achieved using a little circumvention like in [this](#)<sup>78</sup> script.

### 13.1.5 How to Extract Images: Non-PDF Documents

You have basically two options:

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

```
>>> pdfbytes = doc.convertToPDF()
>>> pdf = fitz.open("pdf", pdfbytes)
>>> # 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]
```

### 13.1.6 How to Extract Images: PDF Documents

Like any other “object” in a PDF, embedded 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. The following assumes you have opened a PDF under the name “doc”:

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. As usual, you can save it as a PNG via method `Pixmap.writePNG()` (or get the corresponding binary data `Pixmap.getPNGData()`). There is no way to tell which image format the embedded original has.
2. Extract the image with instruction `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 the image. 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 cross reference numbers ‘xref’ of images?”. There are two answers to this:

- a. “Inspect the page objects” Loop through the document’s page number list and execute `Document.getPageImageList()` for each page number. The result is a list of list, and its items look like `[xref, smask, ...]`, containing the xref of an image shown on that page. This xref can then be used with

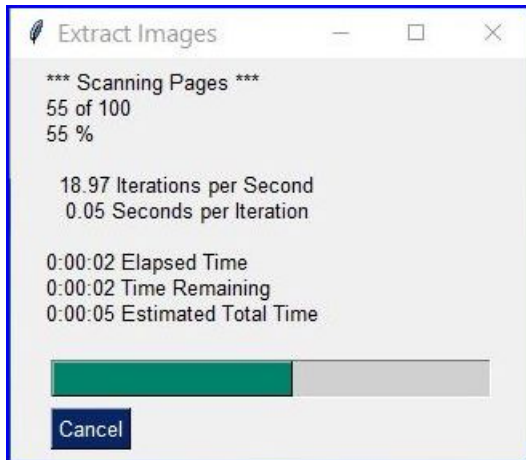
<sup>78</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/show-no-annot.py>

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.

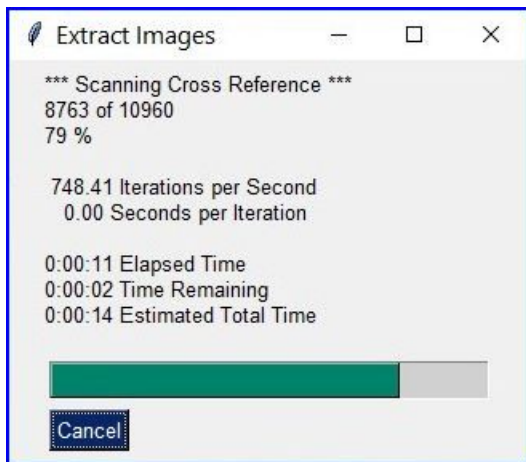
- b. **“No need to know”** Loop through the list of **all xrefs** of the document and perform a `Document.extractImage()` for each one. If the returned dictionary is empty, then continue – this xref is no image. Use this method if the PDF is **damaged (unusable pages)**. Note that a PDF often contains “pseudo-images” (“stencil masks”) with the special purpose to specify 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`<sup>79</sup> extracts images by page:



and `extract-imgb.py`<sup>80</sup> extracts images by cross reference number:



---

### 13.1.7 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.

<sup>79</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/extract-imga.py>

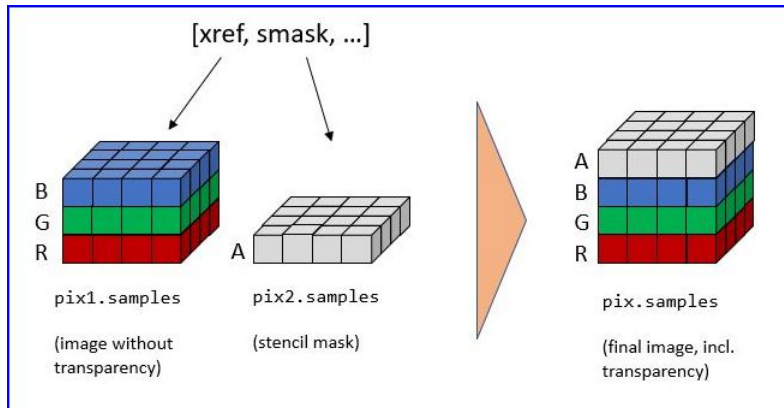
<sup>80</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/extract-imgb.py>

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 cross reference number and `smask`, if positive, is the cross reference number of a stencil mask.
2. The (dictionary) results of `Document.extractImage()` have a key "smask", which also contains any stencil mask's cross reference number 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`<sup>81</sup>, and `extract-imgb.py`<sup>82</sup> above also contain this logic.

### 13.1.8 How to Make one PDF of all your Pictures

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:

```
import os, fitz
import PySimpleGUI as psg                                # for showing 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):
```

(continues on next page)

<sup>81</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/extract-imga.py>

<sup>82</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/extract-imgb.py>

(continued from previous page)

```

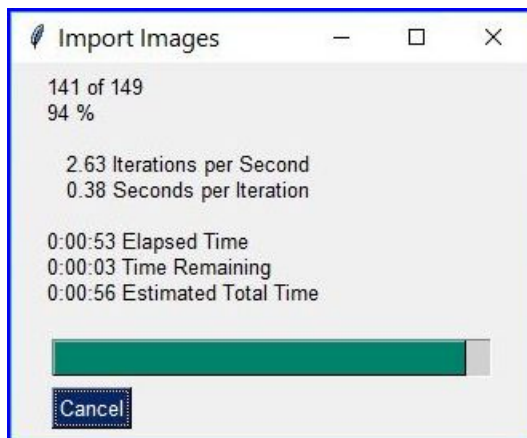
img = fitz.open(os.path.join(imgdir, f)) # open pic as document
rect = img[0].rect                       # pic dimension
pdfbytes = img.convertToPDF()             # make a PDF stream
img.close()                             # no longer needed
imgPDF = fitz.open("pdf", pdfbytes)       # open stream as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

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

```

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

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



Look [here](https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/all-my-pics-inserted.py)<sup>83</sup> for a more complete source code: it offers a directory selection dialog and skips unsupported files and non-file entries.

**Note:** We could 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** the (image) files. You would need a suitable PDF viewer that can display and / or extract embedded files:

```

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

```

(continues on next page)

<sup>83</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/all-my-pics-inserted.py>

(continued from previous page)

```

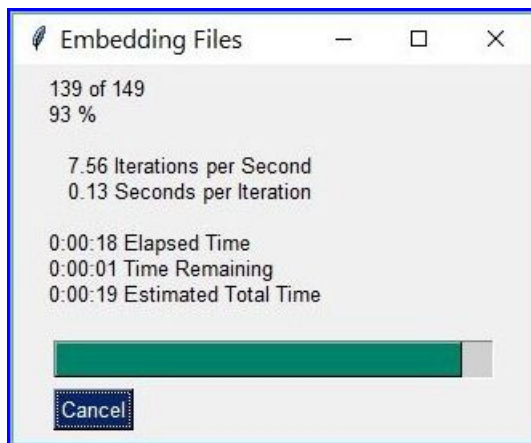
imglist = os.listdir(imgdir)          # list of pictures
imgcount = len(imglist)               # pic count
imglist.sort()                       # nicely sort them

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

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

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

```



This is by far the fastest method, and it also produces the smallest possible output file size. The above pictures needed 20 seconds on my machine and yielded a PDF size of 510 MB. Look [here](#)<sup>84</sup> for a more complete source code: it offers a directory selection dialog and skips non-file entries.

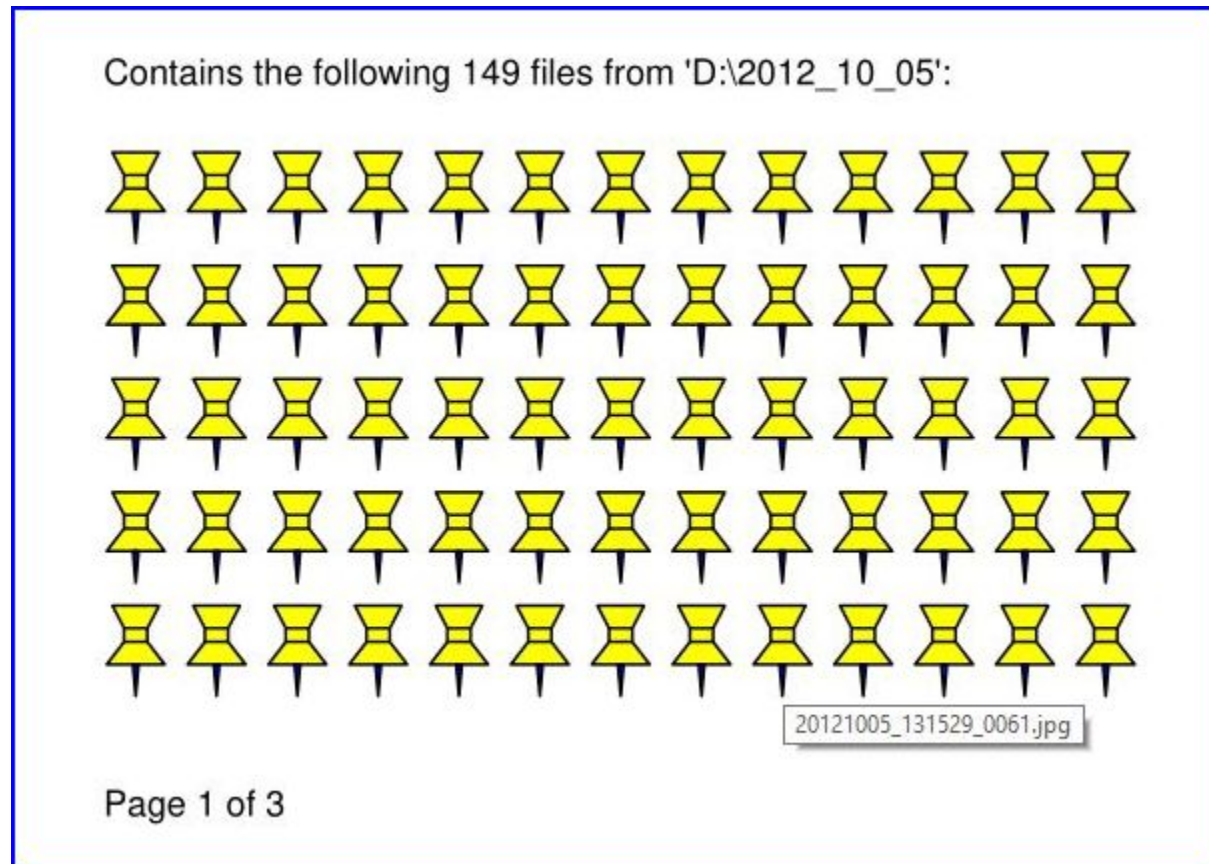
### Method 3: Attaching Files

A third way to achieve this task is **attaching files** via page annotations see [here](#)<sup>85</sup> for the complete source code.

This has a similar performance as the previous script and it also produces a similar file size. In this example, we have chosen a small page size to demonstrate the automatic generation of “protocol” pages as necessary. Here is the first page:

<sup>84</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/all-my-pics-embedded.py>

<sup>85</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/all-my-pics-attached.py>



---

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

---

---

**Note:** We strongly recommend using the awesome package [PySimpleGUI](https://pypi.org/project/PySimpleGUI/)<sup>86</sup> 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!

---

### 13.1.9 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 increased 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 embedded raster graphic elements.

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

---

<sup>86</sup> <https://pypi.org/project/PySimpleGUI/>

## 13.2 Text

### 13.2.1 How to Extract all Document Text

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

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

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

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

The output will be plain text as it is coded in the document. No effort is made to prettify in any way. 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.getTextBlocks()`. 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.getTextWords()`. Its items are words with position information. Use it to determine text contained in a given rectangle – see next section.

### 13.2.2 How to Extract Text from within a Rectangle

Please refer to the script `textboxtract.py`<sup>87</sup>.

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

<sup>87</sup> <https://github.com/JorjMcKie/PyMuPDF-Utilities/blob/master/textboxtract.py>



sion. Radiometrische Untersuchungen lieferten für alle dasselbe Alter: 3,95 Milliarden Jahre. Ein Team des California Institute of Technology in Pasadena bestätigte den Befund kurz darauf.

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

Doch von Anfang an war dieses Szenario umstritten, vor allem wegen der nicht eindeutigen Datierung des Gesteins. Die Altersbestimmung basierte in erster Linie auf dem Verhältniss von Argon-40 und Kalium-40. Letzteres ist radioaktiv und zerfällt mit einer Halbwertszeit von 1,25 Milliarden Jahren in stabile Argon-40. Die Kalium-40-Tenoren

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

Select the words strictly contained in rectangle

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

Or, more forgiving, respectively:

Select the words intersecting the rectangle

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

---

### 13.2.3 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 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).



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](#)<sup>88</sup> and then use the full page rectangle.

### 13.2.4 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](#)<sup>89</sup> strives to exactly do that. You may want to have a look at it and adjust it to your liking.

### 13.2.5 How to Search for and Mark Text

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

This method has advantages and drawbacks. Pros are

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

Disadvantages:

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

But you have other options:

```
import sys
import fitz

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

(continues on next page)

<sup>88</sup> <https://github.com/rk700/PyMuPDF/wiki/How-to-extract-text-from-a-rectangle>

<sup>89</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/wxTableExtract.py>

(continued from previous page)

```

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 [GitHub](https://github.com/rk700/PyMuPDF)<sup>3</sup>. We also are registered on [PyPI](https://pypi.org/project/PyMuPDF/)<sup>4</sup>.

For MS Windows and popular Python versions on Mac OSX and Linux we have created a script that 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:

<sup>1</sup> <http://www.mupdf.com/>

<sup>2</sup> <http://www.sumatrapdfreader.org/>

<sup>3</sup> <https://github.com/rk700/PyMuPDF>

<sup>4</sup> <https://pypi.org/project/PyMuPDF/>

### 13.2.6 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 shearing or mirroring,
  - independently from morphing and in addition to that, text can be rotated by multiples of 90 degrees.

All of the above is provided by three basic [Page](#) methods:

- [Page.insertFont\(\)](#) to install a font referencable by the page. This can be a new font (e.g. provided as a file), a font already present somewhere in this or another PDF, or a built-in font.
- [Page.insertText\(\)](#) to write some text simply line by line. This uses a [Shape](#) method with the same name which provides additional options.
- [Page.insertTextbox\(\)](#) to 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. This method invokes a corresponding method of [Shape](#) as well.

## How to Output 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 = sitz.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 output **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. Line **width is not measured** here.

The height of each line is calculated as  $1.2 * \text{fontsize}$ .

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 set to achieve the desired result:

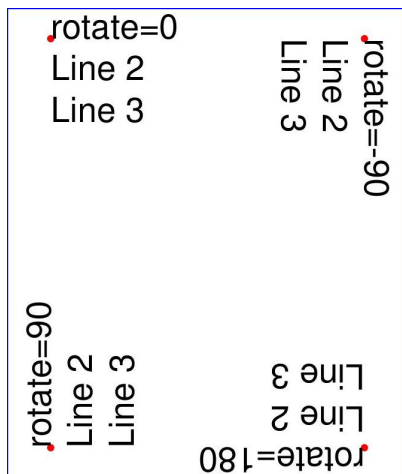
```
import fitz
doc = fitz.open()
page = doc.newPage()
# the text strings, each having 3 lines
text1 = "rotate=0\nLine 2\nLine 3"
text2 = "rotate=90\nLine 2\nLine 3"
text3 = "rotate=-90\nLine 2\nLine 3"
text4 = "rotate=180\nLine 2\nLine 3"
red = (1, 0, 0) # the color for the red dots
# the insertion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

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

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

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

This is the result:



---

### How to Fill a Text Box

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

```

import fitz
doc = fitz.open(...)                # new or existing PDF
page = doc.newPage()                # new page, or choose doc[n]
r1 = fitz.Rect(50,100,100,150)      # a 50x50 rectangle
disp = fitz.Rect(55, 0, 55, 0)      # add this to get more rects
r2 = r1 + disp                      # 2nd rect
r3 = r1 + disp * 2                  # 3rd rect
r4 = r1 + disp * 3                  # 4th rect
t1 = "text with rotate = 0."        # the texts we will put in
t2 = "text with rotate = 90."
t3 = "text with rotate = -90."
t4 = "text with rotate = 180."

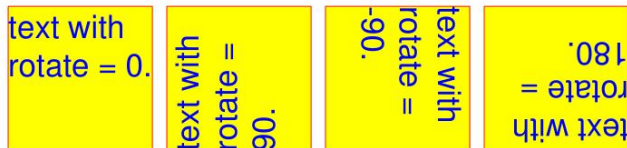
red = (1,0,0)                       # some colors
gold = (1,1,0)
blue = (0,0,1)

"""We use a Shape object (something like a canvas) to output the text and
the rectangles surrounding it for demonstration.
"""

img = page.newShape()               # create Shape
img.drawRect(r1)                    # draw rectangles
img.drawRect(r2)                    # giving them
img.drawRect(r3)                    # a yellow background
img.drawRect(r4)                    # and a red border
img.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 = img.insertTextbox(r1, t1, color = blue)
rc = img.insertTextbox(r2, t2, color = blue, rotate = 90)
rc = img.insertTextbox(r3, t3, color = blue, rotate = -90)
rc = img.insertTextbox(r4, t4, color = blue, rotate = 180)
img.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:



## 13.3 Annotations

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

- New annotation type support for ‘Ink’, ‘Rubber Stamp’ and ‘Squiggly’ annotations. Ink annots simulate handwritings by combining one or more lists of interconnected points. Stamps are intended to visually inform about a document’s status or intended usage (like “draft”, “confidential”, etc.). ‘Squiggly’ is a text marker annot, which underlines selected text with a zigzagged line.
- **Extended ‘FreeText’ support:**
  1. all characters from the Latin character set are now available,

2. colors of text, rectangle background and rectangle border can be independently set
  3. text in rectangle can be rotated by either +90 or -90 degrees
  4. text is automatically wrapped (made multi-line) in available rectangle
  5. all Base-14 fonts are now available (*normal* variants only, i.e. no bold, no italic).
- MuPDF now supports line end icons for 'Line' annots (only). PyMuPDF supported that in v1.13.x already – and for (almost) the full range of applicable types. So we adjusted the appearance of 'Polygon' and 'PolyLine' annots to closely resemble the one of MuPDF for 'Line'.
  - MuPDF now provides its own annotation icons where relevant. PyMuPDF switched to using them (for 'FileAttachment' and 'Text' ["sticky note"] so far).
  - MuPDF now also supports 'Caret', 'Movie', 'Sound' and 'Signature' annotations, which we may include in PyMuPDF at some later time.

### 13.3.1 How to Add and Modify Annotations

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

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

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

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

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

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

doc = fitz.open()
```

(continues on next page)

(continued from previous page)

```

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

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

annot = page.addTextAnnot(r.tl, t1)
annot.setColors(colors2)
annot.update()
print_descr(annot.rect, annot)
print("added 'Sticky Note'")

pos = annot.rect.tl + displ.tl

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

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

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

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

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

r+= displ
annot = page.addPolygonAnnot([r.bl, r.tr, r.br, r.tl])
annot.setBorder(border)
annot.setColors(colors)
annot.setLineEnds(fitz.ANNOT_LE_Diamond, fitz.ANNOT_LE_Circle)
annot.update()

```

(continues on next page)

(continued from previous page)

```
print_descr(annot.rect, annot)
print("added 'Polygon'")

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

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

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

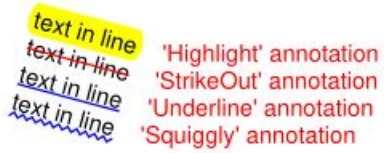
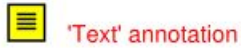
r+= displ
annot = page.addFileAnnot(r.tl, b"just anything for testing", "testdata.txt")
annot.setColors(colors2)
annot.update()
print_descr(annot.rect, annot)
print("added 'FileAttachment'")

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

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

This script should lead to the following output:





### 13.3.2 How to Mark Text

This script searches for text and marks it:

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

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

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

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

(continues on next page)

(continued from previous page)

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

# loop through the found locations to add a marker
for r in rl:
    page.addSquigglyAnnot(r)

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

The result looks like this:



---

### 13.3.3 How to Use FreeText

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

r3 = r2 + (0,75,0,75)

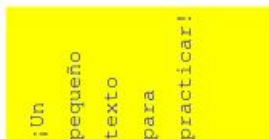
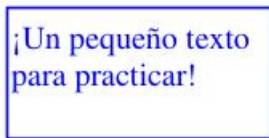
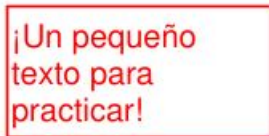
# the text, Latin alphabet
t = "¡Un pequeño texto para practicar!"

# 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":0.0})
a3.update(fontsize=8, fill_color=gold)

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

```

The result looks like this:



### 13.3.4 How to Use Ink Annotations

Ink annotations are used to contain freehand scribbling. Technically an ink annotation is implemented as a list of list of points. Each list of points is regarded as a continuous line connecting them. Different point lists represent disconnected line segments of the scribbling.

The following script creates two line segments and puts them in a given rectangle. The lines themselves are arbitrarily chosen to be the sine and the cosine function graphs:

```

import math
import fitz

#-----
# preliminary stuff
#-----
w360 = math.pi * 2                                # go through full circle

```

(continues on next page)

(continued from previous page)

```

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 here
for x in range(362):                             # now fill in the values
    x_coord = x * x_step + first_x                # current x coordinate
    y = -math.sin(x * deg)                        # sine
    p = (x_coord, y * y_scale + first_y)          # corresponding point
    sin_points.append(p)                         # append
    y = -math.cos(x * deg)                        # cosine
    p = (x_coord, y * y_scale + first_y)          # corresponding point
    cos_points.append(p)                         # append

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

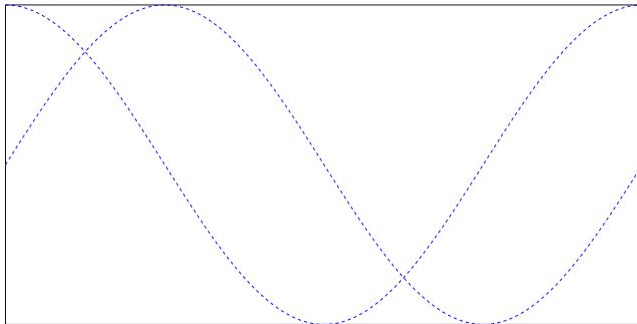
#-----
# add the Ink annotation, consisting of 2 segments
#-----
annot = page.addInkAnnot((sin_points, cos_points))
# let it look a little nicer
annot.setBorder({"width":0.3, "dashes":[1]})      # line thickness, some dashing
annot.setColors({"stroke":(0,0,1)})              # make the lines blue
annot.update()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

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

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

```

This is the result:



## 13.4 General

### 13.4.1 How to Open with a Wrong File Extension

If you have a document which does not have the right 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. Also consult the [Document](#) chapter for a full description.

### 13.4.2 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](#)<sup>90</sup>, [new-annots.py](#)<sup>91</sup>.

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

### 13.4.3 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:

<sup>90</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/embedded-list.py>

<sup>91</sup> <https://github.com/rk700/PyMuPDF/blob/master/demo/new-annots.py>

```
>>> 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:

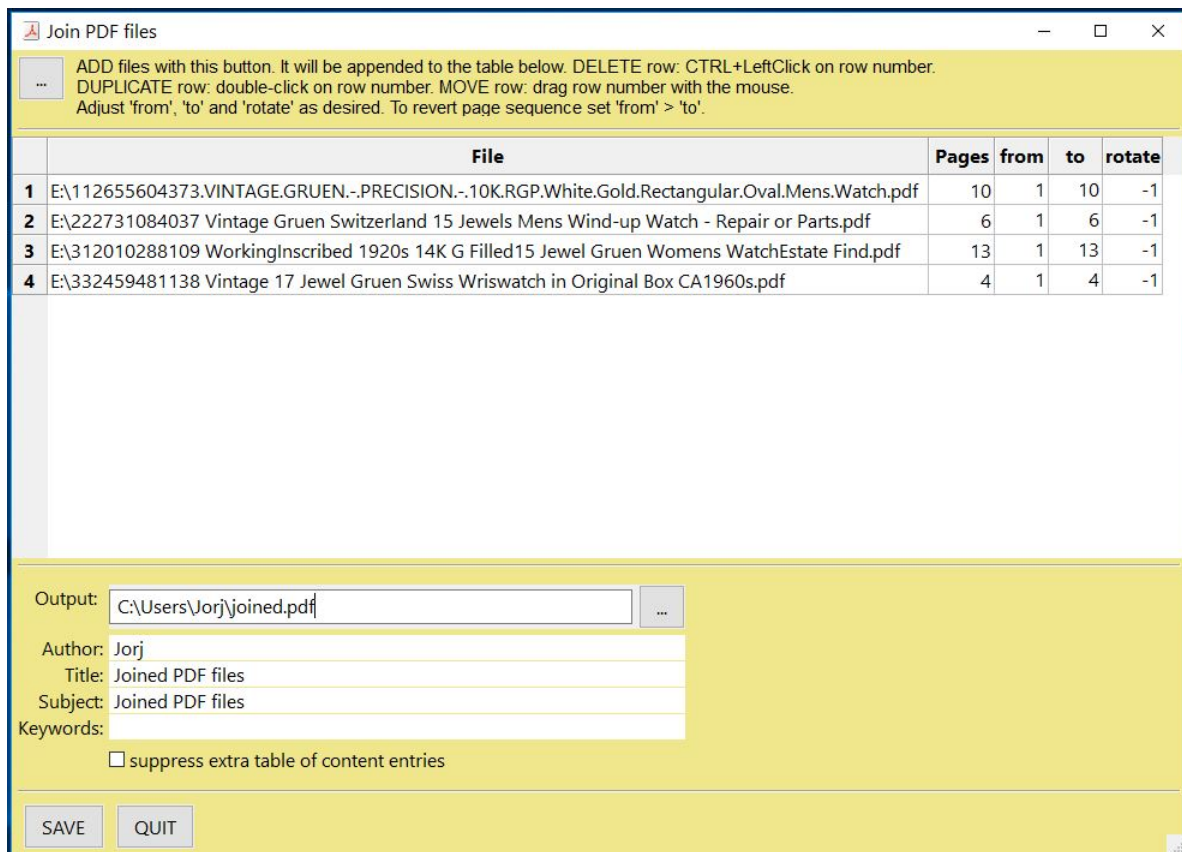
```
>>> doc.select(p_even)    # only the even pages left over
>>> doc.save("even.pdf")  # save the "even" PDF
>>> doc.close()          # recycle the file
>>> doc = fitz.open(doc.name) # re-open
>>> doc.select(p_odd)     # and do the same with the odd pages
>>> doc.save("odd.pdf")
```

For more information also have a look at this Wiki [article](#)<sup>92</sup>.

### 13.4.4 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](#)<sup>93</sup> contains a full description.

The GUI script `PDFjoiner.py`<sup>94</sup> uses this method to join a list of files while also joining the respective table of contents segments. It looks like this:



<sup>92</sup> <https://github.com/rk700/PyMuPDF/wiki/Rearranging-Pages-of-a-PDF>

<sup>93</sup> <https://github.com/rk700/PyMuPDF/wiki/Inserting-Pages-from-other-PDFs>

<sup>94</sup> <https://github.com/rk700/PyMuPDF/blob/master/examples/PDFjoiner.py>

### 13.4.5 How to Add Pages

There are 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.

### 13.4.6 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`<sup>95</sup> is used here as an example).

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

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

#-----
# 'Tolerant' PDF reader
#-----
def reader(fname, password = None):
    idata = open(fname, "rb").read()      # read the PDF into memory and
    ibuffer = BytesIO(idata)             # convert to stream
    if password is None:
        try:
            return PdfReader(ibuffer)     # if this works: fine!
        except:
            pass
    del ibuffer                          # free some storage
    # either we need a password or it is a problem-PDF
    # create a repaired / decompressed / decrypted version
    doc = fitz.open("pdf", idata)
    if password is not None:             # decrypt if password provided
        rc = doc.authenticate(password)
        if not rc > 0:
            raise ValueError("wrong password")
    c = doc.write(garbage=3, deflate=True)
    del doc                              # close & delete doc
    return PdfReader(BytesIO(c))         # let pdfrw retry
#-----
# Main program
#-----
pdf = reader("pymupdf.pdf", password = None) # include a password if necessary
print pdf.Info
# do further processing
```

With the command line utility `pdftk` (available<sup>96</sup> for Windows only, but reported to also run under `Wine`<sup>97</sup>) a similar result can be achieved, see [here](#)<sup>98</sup>. However, you must invoke it as a separate process via `subprocess.Popen`, using `stdin` and `stdout` as communication vehicles.

### 13.4.7 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:

---

<sup>95</sup> <https://pypi.python.org/pypi/pdfrw>

<sup>96</sup> <https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/>

<sup>97</sup> <https://www.winehq.org/>

<sup>98</sup> <http://www.overthere.co.uk/2013/07/22/improving-pypdf2-with-pdftk/>



```

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

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

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

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

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

Dependencies
-----
PyMuPDF 1.12.2 or later
'''

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

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

    #-----
    # example: cut input page into 2 x 2 parts
    #-----
    r1 = r * 0.5                # top left rect
    r2 = r1 + (r1.width, 0, r1.width, 0) # top right rect
    r3 = r1 + (0, r1.height, 0, r1.height) # bottom left rect
    r4 = fitz.Rect(r1.br, r.br) # bottom right rect
    rect_list = [r1, r2, r3, r4] # put them in a list

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

# that's it, save output file

```

(continues on next page)

(continued from previous page)

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

### 13.4.8 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]
```

(continues on next page)

(continued from previous page)

```

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

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

```

### 13.4.9 How to Convert Any Document to PDF

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

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

```

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

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

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

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

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

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

```

(continues on next page)

(continued from previous page)

```

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

# now print any MuPDF warnings or errors:
errors = fitz.TOOLS.fitz_stderr
if errors:                       # any issues?
    print(errors)
    fitz.TOOLS.fitz_stderr_reset() # empty the message store

```

### 13.4.10 How to Access Messages Issued by MuPDF

For motivation and some theory background see [Redirecting Error and Warning Messages](#). Since v1.14.0 we intercept warning and error messages by MuPDF so they no longer appear on the operating system's standard output devices STDOUT, STDERR.

These messages can be safely ignored in many cases, but occasionally do serve diagnostic purposes, e.g. when a corrupted document has been opened.

The messages are not necessarily pertaining to any specific document, so we keep them in an independent store as a string object, accessible via the [Tools](#) class. Every new message is appended to any existing ones, separated by a newline character.

Here is an interactive session making use of this message store:

```
>>> import fitz
>>> doc = fitz.open("acronis.xps") # some XPS document
>>> fitz.TOOLS.fitz_stderr          # message store is still empty
''
>>> pdfbytes = doc.convertToPDF()   # convert XPS to PDF
>>> fitz.TOOLS.fitz_stderr          # and look at the message store:
u'warning: freetype getting character advance: invalid glyph index\n'
>>> fitz.TOOLS.fitz_stderr_reset()  # empty the message store
>>> fitz.TOOLS.fitz_stderr          # and show it worked
''
>>> doc.close()                    # try another document: SVG this time
>>> doc = fitz.open("acronis.svg")
>>> fitz.TOOLS.fitz_stderr          # still no complaints?
''
>>> pdfbytes = doc.convertToPDF()   # convert that one too
>>> fitz.TOOLS.fitz_stderr          # and see what would have gone to system STDERR
'warning: ... repeated 3 times ...\nwarning: push viewport: 0 0 594.75 841.5\nwarning: push
↳viewbox: 0 0 594.75 841.5\nwarning: push viewport: 0 0 594.75 841.5\nwarning: ... repeated 2
↳times ...\nwarning: push viewport: 0 0 980 71\nwarning: push viewport: 0 0 594.75 841.
↳5\nwarning: ... repeated 2512 times ...\nwarning: push viewport: 0 0 112 33\nwarning: push
↳viewport: 0 0 594.75 841.5\nwarning: ... repeated 2 times ...\nwarning: push viewport: 0 0 181
↳120\nwarning: push viewport: 0 0 94 54\nwarning: ... repeated 2 times ...\nwarning: push
↳viewport: 0 0 130 88\nwarning: ... repeated 2 times ...\nwarning: push viewport: 0 0 181
↳115\nwarning: push viewport: 0 0 594.75 841.5\n'
>>>
```

## 13.5 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.

### 13.5.1 How to Iterate through the XREF

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

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

A PDF object definition is an ordinary ASCII string.

---

### 13.5.2 How to Handle Object Streams

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

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

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

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

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

---

### 13.5.3 How to Handle Page Contents

Every PDF page has one or more `/Contents` objects. 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 Reference 1.7](#).

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

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

There are good technical arguments for having multiple `/Contents` objects:

- It is a lot easier and faster to just add new `/Contents` objects than maintaining a single big one (which entails reading, decompressing, modifying, recompressing, and rewriting it each time).
- 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: self-concatenate multiple contents
>>> for page in doc:
    cont = b""              # initialize contents
    for xref in page._getContents(): # loop through content xrefs
        cont += doc._getXrefStream(xref)
    # do something with the combined contents

```

The clean function `Page._cleanContents()` does a lot more than just glueing `/Contents` objects: it also corrects the PDF operator syntax of the page and also that of **all of its annotations** (each *Annot* annotation also has its own contents object!).

And of course, `Page._cleanContents()` writes back its results to the PDF: when saving it, it will reflect those changes. The same happens for the complete PDF when you use the `clean=True` parameter in `Document.save()`.

This may exceed what you actually wanted to achieve.

### 13.5.4 How to Access the PDF Catalog Object

This is a central (“root”) object of a PDF which serves as a starting point to reach other objects and which contains any global options for the PDF:

```

>>> import fitz
>>> doc=fitz.open("PyMuPDF.pdf")
>>> cat = doc._getPDFRoot()           # get xref of the /Catalog
>>> print(doc._getXrefString(cat))     # print object definition
<<
  /Type/Catalog           % object type
  /Pages 3593 0 R         % points to page object tree
  /OpenAction 225 0 R     % action to perform on open
  /Names 3832 0 R         % points to global names tree
  /PageMode/UseOutlines   % show the TOC initially
  /PageLabels<</Nums[0<</S/D>>2<</S/r>>8<</S/D>>]>> % names given to pages
  /Outlines 3835 0 R      % points to start of 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 catalogue see section 3.6.1 on page 137 of the *Adobe PDF Reference 1.7*.

### 13.5.5 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
>>> doc._getXrefString(metaxref)                   # object definition
'<</Subtype/XML/Length 3801/Type/Metadata>>'
>>> xmlmetadata = doc._getXrefStream(metaxref)      # XML data (stream - bytes obj)
>>> print(xmlmetadata.decode("utf8"))              # print str version of bytes
<?xpacket begin="\uffeff" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmpdk="3.1-702">
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
...
omitted data
...
<?xpacket end="w"?>
```

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

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



## CHANGE LOGS

### 14.1 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.

### 14.2 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.

### 14.3 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.

## 14.4 Changes in Version 1.13.19

This version contains some technical / performance improvements and bug fixes.

- **Changed** memory management: for Python 3 builds, Python memory management is exclusively used across all C-level code (i.e. no more native `malloc()` in MuPDF code or PyMuPDF interface code). This leads to improved memory usage profiles and also some runtime improvements: we have seen > 2% shorter runtimes for text extractions and pixmap creations (on Windows machines only to date).
- **Fixed** an error occurring in Python 2.7, which crashed the interpreter when using `TextPage.extractRAWDICT()` (= `Page.getText("rawdick")`).
- **Fixed** an error occurring in Python 2.7, when creating link destinations.
- **Extended** the *Collection of Recipes* chapter with more examples.

## 14.5 Changes in Version 1.13.18

- **Added** method `TextPage.extractRAWDICT()`, and a corresponding new string parameter “rawdick” to method `Page.getText()`. It extracts text and images from a page in Python *dict* form like `TextPage.extractDICT()`, but with the detail level of `TextPage.extractXML()`, which is position information down to each single character.

## 14.6 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.

## 14.7 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.

## 14.8 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.

## 14.9 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', 'Poly-Line', '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.

## 14.10 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.

## 14.11 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.

## 14.12 Changes in Version 1.13.11

While the preceeding patch subversions only contained various fixes, this version again introduces major new features:

- **Added** basic support for PDF widget annotations. You can now add PDF form fields of types Text, CheckBox, ListBox and ComboBox. Where necessary, the PDF is tranformed to a Form PDF with the first added widget.
- **Fixed** issues #176 (“wrong file embedding”), #177 (“segment fault when invoking `page.getText()`”) and #179 (“Segmentation fault using `page.getLinks()` on encrypted PDF”).

## 14.13 Changes in Version 1.13.7

- **Added** support of variable page sizes for reflowable documents (e-books, HTML, etc.): new parameters `rect` and `fontsize` in `Document` creation (`open`), and as a separate method `Document.layout()`.
- **Added** `Annot` creation of many annotations types: sticky notes, free text, circle, rectangle, line, polygon, polyline and text markers.
- **Added** support of annotation transparency (`Annot.opacity`, `Annot.setOpacity()`).
- **Changed** `Annot.vertices`: point coordinates are now grouped as pairs of floats (no longer as separate floats).
- **Changed** annotation colors dictionary: the two keys are now named "stroke" (formerly "common") and "fill".
- **Added** `Document.isDirty` which is `True` if a PDF has been changed in this session. Reset to `False` on each `Document.save()` or `Document.write()`.

## 14.14 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.

## 14.15 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.

## 14.16 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.

## 14.17 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.

## 14.18 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).

## 14.19 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.

## 14.20 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](#)<sup>99</sup>. 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`.

---

<sup>99</sup> <https://github.com/rk700/PyMuPDF/issues/156>

- 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](#)<sup>100</sup>.
- `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)<sup>101</sup>.

## 14.21 Changes in Version 1.12.4

This is an extension of 1.12.3.

- Fix of [issue #147](#)<sup>102</sup>: methods `Document.getPageFontlist()` and `Document.getPageImagelist()` now also show fonts and images contained in `/Resources` nested via “Form XObjects”.
- Temporary fix of [issue #148](#)<sup>103</sup>: Saving to new PDF files will now automatically use `garbage = 2` if a lower value is given. Final fix is to be expected with MuPDF's next version. At that point we will remove this circumvention.
- Preventive fix of illegally using stencil / image mask pixmaps in some methods.
- 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.

## 14.22 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).

## 14.23 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.

---

<sup>100</sup> <https://github.com/rk700/PyMuPDF/issues/148>

<sup>101</sup> <https://github.com/rk700/PyMuPDF/issues/133>

<sup>102</sup> <https://github.com/rk700/PyMuPDF/issues/147>

<sup>103</sup> <https://github.com/rk700/PyMuPDF/issues/148>

## 14.24 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.

## 14.25 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.



## 14.26 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.

## 14.27 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 /Contents object.
- All `Page` draw methods now use the new `Shape` class.
- Text insertion methods `insertText()` and `insertTextBox()` now support morphing in addition to text rotation. They have become part of the `Shape` class and thus allow text to be freely combined with graphics.
- A new `Pixmap` constructor allows creating pixmap copies with an added alpha channel. A new method also allows directly manipulating alpha values.
- Binary algebraic operations with geometry objects (matrices, rectangles and points) now generally also support lists or tuples as the second operand. You can add a tuple `(x, y)` of numbers to a `Point`. In this context, such sequences are called “point-like” (resp. matrix-like, rectangle-like).
- Geometry objects now fully support in-place operators. For example, `p /= m` replaces point `p` with `p * 1/m` for a number, or `p * ~m` for a matrix-like object `m`. Similarly, if `r` is a rectangle, then `r |= (3, 4)` is the new rectangle that also includes `fitz.Point(3, 4)`, and `r &= (1, 2, 3, 4)` is its intersection with `fitz.Rect(1, 2, 3, 4)`.

## 14.28 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 be extracted and changed.

## 14.29 Changes in Version 1.10.0

### 14.29.1 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.

### 14.29.2 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.

## 14.30 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.

## 14.31 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`.

## 14.32 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.

- `__init__()`Colorspace method, 62
- `__init__()`Device method, 130
- `__init__()`DisplayList method, 131
- `__init__()`Document method, 18
- `__init__()`IRect method, 75
- `__init__()`Matrix method, 66, 67
- `__init__()`Pixmap method, 53–55
- `__init__()`Point method, 84, 85
- `__init__()`Quad method, 88
- `__init__()`Rect method, 79
- `__init__()`Shape method, 90
- `_cleanContents()`Annot method, 125
- `_cleanContents()`Page method, 125
- `_delXmlMetadata()`Document method, 121
- `_deleteObject()`Document method, 121
- `_getContents()`Page method, 124
- `_getGCTXerrmsg()`Document method, 126
- `_getNewXref()`Document method, 126
- `_getOLRootNumber()`Document method, 127
- `_getPageObjNumber()`Document method, 122
- `_getPageXref()`Document method, 122
- `_getXmlMetadataXref()`Document method, 122
- `_getXrefLength()`Document method, 126
- `_getXrefStream()`Document method, 127
- `_getXrefString()`Document method, 126
- `_setContents()`Page method, 124
- `_updateObject()`Document method, 126
- `_updateStream()`Document method, 127
- `aMatrix` attribute, 68
- `abs_unitPoint` attribute, 86
- `addCircleAnnot()`Page method, 40
- `addFileAnnot()`Page method, 39
- `addFreetextAnnot()`Page method, 38
- `addHighlightAnnot()`Page method, 40
- `addInkAnnot()`Page method, 39
- `addLineAnnot()`Page method, 39
- `addPolygonAnnot()`Page method, 40
- `addPolylineAnnot()`Page method, 40
- `addRectAnnot()`Page method, 40
- `addSquigglyAnnot()`Page method, 40
- `addStampAnnot()`Page method, 41
- `addStrikeoutAnnot()`Page method, 40
- `addTextAnnot()`Page method, 38
- `addUnderlineAnnot()`Page method, 40
- `addWidget()`Page method, 41
- `align`
  - Page.insertTextbox args, 42
  - Shape.insertTextbox args, 95
- `alpha`
  - Annot.getPixmap args, 104
  - DisplayList.getPixmap args, 131
  - Page.getPixmap args, 46
- `alphaPixmap` attribute, 57
- Annotbuilt-in class, 104
- Annot.fileUpd args
  - buffer, 107
  - desc, 107
  - filename, 107
  - ufilename, 107
- Annot.getPixmap args
  - alpha, 104
  - colorspace, 104
  - matrix, 104
- Annot.update args
  - border\_color, 106
  - fill\_color, 106
  - fontsize, 106
  - rotate, 106
  - text\_color, 106
- ANNOT\_3Dbuilt-in variable, 143
- ANNOT\_CARETbuilt-in variable, 143
- ANNOT\_CIRCLEbuilt-in variable, 142
- ANNOT\_FILEATTACHMENTbuilt-in variable, 143
- ANNOT\_FREETEXTbuilt-in variable, 142
- ANNOT\_HIGHLIGHTbuilt-in variable, 142
- ANNOT\_INKbuilt-in variable, 143
- ANNOT\_LE\_Buttbuilt-in variable, 146
- ANNOT\_LE\_Circlebuilt-in variable, 145
- ANNOT\_LE\_ClosedArrowbuilt-in variable, 146
- ANNOT\_LE\_Diamondbuilt-in variable, 145
- ANNOT\_LE\_Nonebuilt-in variable, 145
- ANNOT\_LE\_OpenArrowbuilt-in variable, 146
- ANNOT\_LE\_RClosedArrowbuilt-in variable, 146
- ANNOT\_LE\_ROpenArrowbuilt-in variable, 146

ANNOT\_LE\_Slashbuilt-in variable, 146  
ANNOT\_LE\_Squarebuilt-in variable, 145  
ANNOT\_LINEbuilt-in variable, 142  
ANNOT\_LINKbuilt-in variable, 142  
ANNOT\_MOVIEbuilt-in variable, 143  
ANNOT\_POLYGONbuilt-in variable, 142  
ANNOT\_POLYLINEbuilt-in variable, 142  
ANNOT\_POPUPbuilt-in variable, 143  
ANNOT\_PRINTERMARKbuilt-in variable, 143  
ANNOT\_SCREENbuilt-in variable, 143  
ANNOT\_SOUNDbuilt-in variable, 143  
ANNOT\_SQUAREbuilt-in variable, 142  
ANNOT\_SQUIGGLYbuilt-in variable, 142  
ANNOT\_STAMPbuilt-in variable, 142  
ANNOT\_STRIKEOUTbuilt-in variable, 142  
ANNOT\_TEXTbuilt-in variable, 142  
ANNOT\_TRAPNETbuilt-in variable, 143  
ANNOT\_UNDERLINEbuilt-in variable, 142  
ANNOT\_WATERMARKbuilt-in variable, 143  
ANNOT\_WG\_CHECKBOXbuilt-in variable, 143  
ANNOT\_WG\_COMBOBOXbuilt-in variable, 143  
ANNOT\_WG\_LISTBOXbuilt-in variable, 143  
ANNOT\_WG\_NOT\_WIDGETbuilt-in variable, 143  
ANNOT\_WG\_PUSHBUTTONbuilt-in variable, 143  
ANNOT\_WG\_RADIOBUTTONbuilt-in variable, 143  
ANNOT\_WG\_SIGNATUREbuilt-in variable, 143  
ANNOT\_WG\_TEXTbuilt-in variable, 143  
ANNOT\_WIDGETbuilt-in variable, 143  
ANNOT\_XF\_Hiddenbuilt-in variable, 144  
ANNOT\_XF\_Invisiblebuilt-in variable, 144  
ANNOT\_XF\_Lockedbuilt-in variable, 144  
ANNOT\_XF\_LockedContentsbuilt-in variable, 144  
ANNOT\_XF\_NoRotatebuilt-in variable, 144  
ANNOT\_XF\_NoViewbuilt-in variable, 144  
ANNOT\_XF\_NoZoombuilt-in variable, 144  
ANNOT\_XF\_Printbuilt-in variable, 144  
ANNOT\_XF\_ReadOnlybuilt-in variable, 144  
ANNOT\_XF\_ToggleNoViewbuilt-in variable, 144  
annotation  
    suppress, 176  
attach  
    embed file, 197  
authenticate()Document method, 19  
  
bMatrix attribute, 68  
Base14\_Fontsbuilt-in variable, 139  
blIRect attribute, 76  
blRect attribute, 81  
borderAnnot attribute, 109  
borderLink attribute, 63  
border\_color  
    Annot.update args, 106  
border\_colorWidget attribute, 110  
border\_dashesWidget attribute, 111

border\_styleWidget attribute, 110  
border\_widthWidget attribute, 111  
bottom\_leftIRect attribute, 76  
bottom\_leftRect attribute, 81  
bottom\_rightIRect attribute, 76  
bottom\_rightRect attribute, 81  
bound()Page method, 38  
brIRect attribute, 76  
brRect attribute, 81  
buffer  
    Annot.fileUpd args, 107  
button\_captionWidget attribute, 111  
  
cMatrix attribute, 68  
choice\_valuesWidget attribute, 111  
clearWith()Pixmap method, 55  
clip  
    DisplayList.getPixmap args, 131  
    Page.getPixmap args, 46  
    Page.showPDFpage args, 47  
close()Document method, 30  
closePath  
    Page.drawBezier args, 43  
    Page.drawCircle args, 42  
    Page.drawCurve args, 43  
    Page.drawLine args, 42  
    Page.drawOval args, 42  
    Page.drawPolyline args, 42  
    Page.drawRect args, 43  
    Page.drawSector args, 42  
    Page.drawSquiggle args, 42  
    Page.drawZigzag args, 42  
    Shape.finish args, 96  
color  
    Document.insertPage args, 26  
    Page.addFreetextAnnot args, 38  
    Page.drawBezier args, 43  
    Page.drawCircle args, 42  
    Page.drawCurve args, 43  
    Page.drawLine args, 42  
    Page.drawOval args, 42  
    Page.drawPolyline args, 42  
    Page.drawRect args, 43  
    Page.drawSector args, 42  
    Page.drawSquiggle args, 42  
    Page.drawZigzag args, 42  
    Page.insertText args, 42  
    Page.insertTextbox args, 42  
    Shape.finish args, 96  
    Shape.insertText args, 95  
    Shape.insertTextbox args, 95  
colorsAnnot attribute, 109  
colorsLink attribute, 63  
colorspace



- Annot.getPixmap args, 104
- DisplayList.getPixmap args, 131
- Page.getPixmap args, 46
- Colorspacebuilt-in class, 62
- colorspacePixmap attribute, 57
- commit()Shape method, 98
- concat()Matrix method, 68
- contains()IRect method, 76
- contains()Rect method, 81
- contentsShape attribute, 99
- ConversionHeader(), 121
- ConversionTrailer(), 121
- convertToPDF, 177
- convertToPDF()Document method, 19
- copyPage()Document method, 28
- copyPixmap()Pixmap method, 56
- CropBoxPage attribute, 49
- CropBoxPositionPage attribute, 49
- CS\_CMYKbuilt-in variable, 139
- CS\_GRAYbuilt-in variable, 139
- CS\_RGBbuilt-in variable, 139
- csCMYKbuilt-in variable, 139
- csGRAYbuilt-in variable, 139
- csRGBbuilt-in variable, 139
- dMatrix attribute, 68
- dashes
  - Page.drawBezier args, 43
  - Page.drawCircle args, 42
  - Page.drawCurve args, 43
  - Page.drawLine args, 42
  - Page.drawOval args, 42
  - Page.drawPolyline args, 42
  - Page.drawRect args, 43
  - Page.drawSector args, 42
  - Page.drawSquiggle args, 42
  - Page.drawZigzag args, 42
  - Shape.finish args, 96
- delete
  - pages, 197
- deleteAnnot()Page method, 41
- deleteLink()Page method, 41
- deletePage()Document method, 27
- deletePageRange()Document method, 27
- desc
  - Annot.fileUpd args, 107
  - Document.embeddedFileAdd args, 28
  - Document.embeddedFileUpd args, 29
- destLink attribute, 64
- destlinkDest attribute, 64
- destOutline attribute, 36
- Devicebuilt-in class, 130
- DisplayListbuilt-in class, 131
- DisplayList.getPixmap args
  - alpha, 131
  - clip, 131
  - colorspace, 131
  - matrix, 131
- distance\_to()Point method, 85
- docShape attribute, 99
- Document
  - open, 18
- Documentbuilt-in class, 18
- Document args
  - filename, 18
  - filetype, 18
  - fontsize, 18
  - rect, 18
  - stream, 18
- Document.convertToPDF args
  - from\_page, 19
  - rotate, 19
  - to\_page, 19
- Document.embeddedFileAdd args
  - desc, 28
  - filename, 28
  - ufilename, 28
- Document.embeddedFileUpd args
  - desc, 29
  - filename, 29
  - ufilename, 29
- Document.insertPage args
  - color, 26
  - fontfile, 26
  - fontname, 26
  - fontsize, 26
  - height, 26
  - width, 26
- Document.insertPDF args
  - from\_page, 25
  - links, 25
  - rotate, 25
  - start\_at, 25
  - to\_page, 25
- Document.layout args
  - fontsize, 23
  - height, 23
  - rect, 23
  - width, 23
- Document.newPage args
  - height, 27
  - width, 27
- downOutline attribute, 35
- drawBezier()Page method, 43
- drawBezier()Shape method, 92
- drawCircle()Page method, 42
- drawCircle()Shape method, 93
- drawCurve()Page method, 43

- drawCurve()Shape method, 93
- drawLine()Page method, 42
- drawLine()Shape method, 90
- drawOval()Page method, 42
- drawOval()Shape method, 93
- drawPolyline()Page method, 42
- drawPolyline()Shape method, 92
- drawRect()Page method, 43
- drawRect()Shape method, 94
- drawSector()Page method, 42
- drawSector()Shape method, 94
- drawSquiggle()Page method, 42
- drawSquiggle()Shape method, 90
- drawZigzag()Page method, 42
- drawZigzag()Shape method, 92
- eMatrix attribute, 68
- embed
  - file, attach, 197
- embed
  - PDF, picture, 179
- embeddedFileAdd, 182
- embeddedFileAdd()Document method, 28
- embeddedFileCountDocument attribute, 31
- embeddedFileDel()Document method, 28
- embeddedFileGet()Document method, 28
- embeddedFileInfo()Document method, 29
- embeddedFileSetInfo()Document method, 29
- embeddedFileUpd()Document method, 29
- even\_odd
  - Shape.finish args, 96
- expandtabs
  - Page.insertTextbox args, 42
  - Shape.insertTextbox args, 95
- extract
  - table, 185
- extract
  - image non-PDF, 177
  - image PDF, 177
  - text rectangle, 183
- extractDICT()TextPage method, 132
- extractFont()Document method, 129
- extractHTML()TextPage method, 132
- extractImage, 177
- extractImage()Document method, 127
- extractJSON()TextPage method, 133
- extractRAWDICT()TextPage method, 133
- extractTEXT()TextPage method, 132
- extractText()TextPage method, 132
- extractXHTML()TextPage method, 133
- extractXML()TextPage method, 133
- fMatrix attribute, 68
- field\_flagsWidget attribute, 111
- field\_nameWidget attribute, 111
- field\_typeWidget attribute, 111
- field\_type\_stringWidget attribute, 111
- field\_valueWidget attribute, 111
- file
  - attach embed, 197
- file extension
  - wrong, 197
- fileGet()Annot method, 106
- fileInfo()Annot method, 106
- filename
  - Annot.fileUpd args, 107
  - Document args, 18
  - Document.embeddedFileAdd args, 28
  - Document.embeddedFileUpd args, 29
  - open args, 18
  - Page.insertImage args, 44
- fileSpecLinkDest attribute, 65
- filetype
  - Document args, 18
  - open args, 18
- fileUpd()Annot method, 107
- fill
  - Page.drawBezier args, 43
  - Page.drawCircle args, 42
  - Page.drawCurve args, 43
  - Page.drawLine args, 42
  - Page.drawOval args, 42
  - Page.drawPolyline args, 42
  - Page.drawRect args, 43
  - Page.drawSector args, 42
  - Page.drawSquiggle args, 42
  - Page.drawZigzag args, 42
  - Shape.finish args, 96
- fill\_color
  - Annot.update args, 106
- fill\_colorWidget attribute, 111
- finish()Shape method, 96
- firstAnnotPage attribute, 50
- firstLinkPage attribute, 50
- fitz\_configTools attribute, 113
- fitz\_stderrTools attribute, 115
- fitz\_stderr\_reset()Tools method, 113
- fitz\_stdoutTools attribute, 115
- fitz\_stdout\_reset()Tools method, 113
- flagsAnnot attribute, 107
- flagslinkDest attribute, 65
- fontbuffer
  - Page.insertFont args, 43
- fontfile
  - Document.insertPage args, 26
  - Page.insertFont args, 43
  - Page.insertText args, 42
  - Page.insertTextbox args, 42

- Shape.insertText args, 95
- Shape.insertTextbox args, 95
- FontInfosDocument attribute, 129
- fontname
  - Document.insertPage args, 26
  - Page.addFreetextAnnot args, 38
  - Page.insertFont args, 43
  - Page.insertText args, 42
  - Page.insertTextbox args, 42
  - Shape.insertText args, 95
  - Shape.insertTextbox args, 95
- fontsize
  - Annot.update args, 106
  - Document args, 18
  - Document.insertPage args, 26
  - Document.layout args, 23
  - open args, 18
  - Page.addFreetextAnnot args, 38
  - Page.insertText args, 42
  - Page.insertTextbox args, 42
  - Shape.insertText args, 95
  - Shape.insertTextbox args, 95
- FormFontsDocument attribute, 32
- from\_page
  - Document.convertToPDF args, 19
  - Document.insertPDF args, 25
- fullSector
  - Page.drawSector args, 42
  - Shape.drawSector args, 94
- gammaWith()Pixmap method, 55
- gen\_id()Tools method, 113
- getArea()IRect method, 76
- getArea()Rect method, 81
- getCharWidths()Document method, 125
- getDisplayList()Page method, 124
- getFontList()Page method, 46
- getImageList()Page method, 46
- getLinks()Page method, 42
- getPageFontList()Document method, 22
- getPageImageList()Document method, 21
- getPagePixmap()Document method, 21
- getPageText()Document method, 22
- getPDFnow(), 120
- getPDFstr(), 121
- getPixmap()Annot method, 104
- getPixmap()DisplayList method, 131
- getPixmap()Page method, 46
- getPNGData()Pixmap method, 57
- getRect()IRect method, 75
- getRectArea()IRect method, 75
- getRectArea()Rect method, 81
- getSVGImage()Page method, 46
- getText()Page method, 45
- getTextBlocks()Page method, 122
- getTextPage()DisplayList method, 131
- getTextWords()Page method, 123
- getToC()Document method, 20
- hPixmap attribute, 57
- height
  - Document.insertPage args, 26
  - Document.layout args, 23
  - Document.newPage args, 27
  - open args, 18
- heightIRect attribute, 77
- heightPixmap attribute, 57
- heightQuad attribute, 89
- heightRect attribute, 82
- heightShape attribute, 99
- hit\_max
  - Page.searchFor args, 48
- image
  - resolution, 175
- image
  - non-PDF, extract, 177
  - PDF, extract, 177
  - SVG, vector, 182
- includePoint()Rect method, 80
- includeRect()Rect method, 80
- infoAnnot attribute, 107
- insertFont()Page method, 43, 123
- insertImage, 182
- insertImage()Page method, 44
- insertLink()Page method, 42
- insertPage()Document method, 26
- insertPDF()Document method, 25
- insertText()Page method, 42
- insertText()Shape method, 95
- insertTextbox()Page method, 42
- insertTextbox()Shape method, 95
- interpolatePixmap attribute, 58
- intersect()IRect method, 76
- intersect()Rect method, 80
- intersects()IRect method, 76
- intersects()Rect method, 81
- invert()Matrix method, 68
- invertIRect()Pixmap method, 56
- IRectbuilt-in class, 75
- irectPixmap attribute, 57
- irectRect attribute, 81
- is\_openOutline attribute, 36
- isClosedDocument attribute, 30
- isEmptyIRect attribute, 77
- isEmptyQuad attribute, 89
- isEmptyRect attribute, 82
- isEncryptedDocument attribute, 30

- isExternalLink attribute, [64](#)
- isExternalOutline attribute, [36](#)
- isFormPDFDocument attribute, [30](#)
- isInfiniteIRect attribute, [77](#)
- isInfiniteRect attribute, [82](#)
- isMaplinkDest attribute, [65](#)
- isPDFDocument attribute, [30](#)
- isRectangularQuad attribute, [89](#)
- isRectilinearMatrix attribute, [68](#)
- isReflowableDocument attribute, [30](#)
- isUrilinkDest attribute, [65](#)
  
- keep\_proportion
  - Page.showPDFpage args, [47](#)
- kindlinkDest attribute, [65](#)
  
- lastPointShape attribute, [99](#)
- layout()Document method, [23](#)
- lineEndsAnnot attribute, [108](#)
- Linkbuilt-in class, [63](#)
- LINK\_FLAG\_B\_VALIDbuilt-in variable, [141](#)
- LINK\_FLAG\_FIT\_Hbuilt-in variable, [142](#)
- LINK\_FLAG\_FIT\_Vbuilt-in variable, [142](#)
- LINK\_FLAG\_L\_VALIDbuilt-in variable, [141](#)
- LINK\_FLAG\_R\_IS\_ZOOMbuilt-in variable, [142](#)
- LINK\_FLAG\_R\_VALIDbuilt-in variable, [141](#)
- LINK\_FLAG\_T\_VALIDbuilt-in variable, [141](#)
- LINK\_GOTObuilt-in variable, [141](#)
- LINK\_GOTORbuilt-in variable, [141](#)
- LINK\_LAUNCHbuilt-in variable, [141](#)
- LINK\_NONEbuilt-in variable, [141](#)
- LINK\_URIBuilt-in variable, [141](#)
- linkDestbuilt-in class, [64](#)
- links
  - Document.insertPDF args, [25](#)
- lIQuad attribute, [88](#)
- loadLinks()Page method, [47](#)
- loadPage()Document method, [19](#)
- lRQuad attribute, [88](#)
- lTlinkDest attribute, [65](#)
  
- matrix
  - Annot.getPixmap args, [104](#)
  - DisplayList.getPixmap args, [131](#)
  - Page.getPixmap args, [46](#)
  - Page.getSVGImage args, [46](#)
- Matrixbuilt-in class, [66](#)
- MediaBoxPage attribute, [49](#)
- MediaBoxSizePage attribute, [49](#)
- metadataDocument attribute, [31](#)
- morph
  - Page.drawBezier args, [43](#)
  - Page.drawCircle args, [42](#)
  - Page.drawCurve args, [43](#)
  - Page.drawLine args, [42](#)
- Page.drawOval args, [42](#)
- Page.drawPolyline args, [42](#)
- Page.drawRect args, [43](#)
- Page.drawSector args, [42](#)
- Page.drawSquiggle args, [42](#)
- Page.drawZigzag args, [42](#)
- Page.insertText args, [42](#)
- Page.insertTextbox args, [42](#)
- Shape.finish args, [96](#)
- Shape.insertText args, [95](#)
- Shape.insertTextbox args, [95](#)
- movePage()Document method, [28](#)
  
- nColorspace attribute, [62](#)
- nPixmap attribute, [58](#)
- nameColorspace attribute, [62](#)
- nameDocument attribute, [31](#)
- namedlinkDest attribute, [65](#)
- needsPassDocument attribute, [30](#)
- newPage()Document method, [27](#)
- newShape()Page method, [48](#)
- newWindowlinkDest attribute, [65](#)
- nextAnnot attribute, [107](#)
- nextLink attribute, [64](#)
- nextOutline attribute, [35](#)
- non-PDF
  - extract image, [177](#)
- normalize()IRect method, [76](#)
- normalize()Rect method, [81](#)
- numberPage attribute, [50](#)
  
- opacityAnnot attribute, [107](#)
- open
  - Document, [18](#)
- open args
  - filename, [18](#)
  - filetype, [18](#)
  - fontsize, [18](#)
  - height, [18](#)
  - rect, [18](#)
  - stream, [18](#)
  - width, [18](#)
- openErrCodeDocument attribute, [31](#)
- openErrMsgDocument attribute, [31](#)
- Outlinebuilt-in class, [35](#)
- outlineDocument attribute, [30](#)
- overlay
  - Page.drawBezier args, [43](#)
  - Page.drawCircle args, [42](#)
  - Page.drawCurve args, [43](#)
  - Page.drawLine args, [42](#)
  - Page.drawOval args, [42](#)
  - Page.drawPolyline args, [42](#)
  - Page.drawRect args, [43](#)

- Page.drawSector args, 42
- Page.drawSquiggle args, 42
- Page.drawZigzag args, 42
- Page.insertImage args, 44
- Page.insertText args, 42
- Page.insertTextbox args, 42
- Page.showPDFpage args, 47
- Shape.commit args, 98
- Pagebuilt-in class, 38
- pagelinkDest attribute, 65
- pageOutline attribute, 35
- pageShape attribute, 99
- Page.addFreetextAnnot args
  - color, 38
  - fontname, 38
  - fontsize, 38
  - rect, 38
  - rotate, 38
- Page.drawBezier args
  - closePath, 43
  - color, 43
  - dashes, 43
  - fill, 43
  - morph, 43
  - overlay, 43
  - roundCap, 43
  - width, 43
- Page.drawCircle args
  - closePath, 42
  - color, 42
  - dashes, 42
  - fill, 42
  - morph, 42
  - overlay, 42
  - roundCap, 42
  - width, 42
- Page.drawCurve args
  - closePath, 43
  - color, 43
  - dashes, 43
  - fill, 43
  - morph, 43
  - overlay, 43
  - roundCap, 43
  - width, 43
- Page.drawLine args
  - closePath, 42
  - color, 42
  - dashes, 42
  - fill, 42
  - morph, 42
  - overlay, 42
  - roundCap, 42
  - width, 42
- Page.drawOval args
  - closePath, 42
  - color, 42
  - dashes, 42
  - fill, 42
  - morph, 42
  - overlay, 42
  - roundCap, 42
  - width, 42
- Page.drawPolyline args
  - closePath, 42
  - color, 42
  - dashes, 42
  - fill, 42
  - morph, 42
  - overlay, 42
  - roundCap, 42
  - width, 42
- Page.drawRect args
  - closePath, 43
  - color, 43
  - dashes, 43
  - fill, 43
  - morph, 43
  - overlay, 43
  - roundCap, 43
  - width, 43
- Page.drawSector args
  - closePath, 42
  - color, 42
  - dashes, 42
  - fill, 42
  - fullSector, 42
  - morph, 42
  - overlay, 42
  - roundCap, 42
  - width, 42
- Page.drawSquiggle args
  - closePath, 42
  - color, 42
  - dashes, 42
  - fill, 42
  - morph, 42
  - overlay, 42
  - roundCap, 42
  - width, 42
- Page.drawZigzag args
  - closePath, 42
  - color, 42
  - dashes, 42
  - fill, 42
  - morph, 42
  - overlay, 42

- roundCap, 42
- width, 42
- Page.getPixmap args
  - alpha, 46
  - clip, 46
  - colorspace, 46
  - matrix, 46
- Page.getSVGImage args
  - matrix, 46
- Page.insertFont args
  - fontbuffer, 43
  - fontfile, 43
  - fontname, 43
  - set\_simple, 43
- Page.insertImage args
  - filename, 44
  - overlay, 44
  - pixmap, 44
  - stream, 44
- Page.insertText args
  - color, 42
  - fontfile, 42
  - fontname, 42
  - fontsize, 42
  - morph, 42
  - overlay, 42
  - rotate, 42
- Page.insertTextbox args
  - align, 42
  - color, 42
  - expandtabs, 42
  - fontfile, 42
  - fontname, 42
  - fontsize, 42
  - morph, 42
  - overlay, 42
  - rotate, 42
- Page.searchFor args
  - hit\_max, 48
- Page.setRotation args
  - rotate, 47
- Page.showPDFpage args
  - clip, 47
  - keep\_proportion, 47
  - overlay, 47
  - reuse\_xref, 47
- pageCountDocument attribute, 31
- pages
  - delete, 197
  - rearrange, 197
- PaperRect(), 120
- PaperSize(), 120
- paperSizes, 120
- parentAnnot attribute, 107
- parentPage attribute, 50
- Partial Pixmaps, 176
- PDF
  - extract image, 177
  - picture embed, 179
- permissionsDocument attribute, 30
- picture
  - embed PDF, 179
- pixmap
  - Page.insertImage args, 44
- Pixmapbuilt-in class, 53
- Pointbuilt-in class, 84
- preRotate()Matrix method, 67
- preScale()Matrix method, 67
- preShear()Matrix method, 67
- preTranslate()Matrix method, 67
- Quadbuilt-in class, 87
- quadIRect attribute, 77
- quadRect attribute, 82
- rblinkDest attribute, 65
- reading order
  - text, 184
- rearrange
  - pages, 197
- rect
  - Document args, 18
  - Document.layout args, 23
  - open args, 18
  - Page.addFreetextAnnot args, 38
- rectAnnot attribute, 107
- Rectbuilt-in class, 79
- rectDisplayList attribute, 132
- rectLink attribute, 63
- rectPage attribute, 50
- rectQuad attribute, 88
- rectShape attribute, 99
- rectWidget attribute, 111
- rectangle
  - extract text, 183
- resolution
  - image, 175
- resolution
  - zoom, 176
- reuse\_xref
  - Page.showPDFpage args, 47
- rotate
  - Annot.update args, 106
  - Document.convertToPDF args, 19
  - Document.insertPDF args, 25
  - Page.addFreetextAnnot args, 38
  - Page.insertText args, 42
  - Page.insertTextbox args, 42

- Page.setRotation args, 47
  - Shape.insertText args, 95
  - Shape.insertTextbox args, 95
- rotationPage attribute, 49
- round()Rect method, 80
- roundCap
  - Page.drawBezier args, 43
  - Page.drawCircle args, 42
  - Page.drawCurve args, 43
  - Page.drawLine args, 42
  - Page.drawOval args, 42
  - Page.drawPolyline args, 42
  - Page.drawRect args, 43
  - Page.drawSector args, 42
  - Page.drawSquiggle args, 42
  - Page.drawZigzag args, 42
  - Shape.finish args, 96
- run()DisplayList method, 131
- run()Page method, 122
- samplesPixmap attribute, 57
- save()Document method, 24
- saveIncr()Document method, 25
- search()TextPage method, 133
- searchFor()Page method, 48
- searchPageFor()Document method, 25
- select()Document method, 23
- set\_simple
  - Page.insertFont args, 43
- setAlpha()Pixmap method, 56
- setBorder()Annot method, 105
- setBorder()Link method, 63
- setColors()Annot method, 105
- setCropBox()Page method, 49
- setFlags()Annot method, 105
- setInfo()Annot method, 104
- setLineEnds()Annot method, 104
- setMetadata()Document method, 23
- setOpacity()Annot method, 105
- setRect()Annot method, 105
- setRotation()Page method, 47
- setToC()Document method, 23
- Shapebuilt-in class, 90
- Shape.commit args
  - overlay, 98
- Shape.drawSector args
  - fullSector, 94
- Shape.finish args
  - closePath, 96
  - color, 96
  - dashes, 96
  - even\_odd, 96
  - fill, 96
  - morph, 96
  - roundCap, 96
  - width, 96
- Shape.insertText args
  - color, 95
  - fontfile, 95
  - fontname, 95
  - fontsize, 95
  - morph, 95
  - rotate, 95
- Shape.insertTextbox args
  - align, 95
  - color, 95
  - expandtabs, 95
  - fontfile, 95
  - fontname, 95
  - fontsize, 95
  - morph, 95
  - rotate, 95
- showPDFpage, 182
  - insertImage;embeddedFileAdd, 179
- showPDFpage()Page method, 47
- shrink()Pixmap method, 55
- sizePixmap attribute, 57
- STAMP\_Approvedbuilt-in variable, 145
- STAMP\_AsIsbuilt-in variable, 145
- STAMP\_Confidentialbuilt-in variable, 145
- STAMP\_Departmentalbuilt-in variable, 145
- STAMP\_Draftbuilt-in variable, 145
- STAMP\_Experimentalbuilt-in variable, 145
- STAMP\_Expiredbuilt-in variable, 145
- STAMP\_Finalbuilt-in variable, 145
- STAMP\_ForCommentbuilt-in variable, 145
- STAMP\_ForPublicReleasebuilt-in variable, 145
- STAMP\_NotApprovedbuilt-in variable, 145
- STAMP\_NotForPublicReleasebuilt-in variable, 145
- STAMP\_Soldbuilt-in variable, 145
- STAMP\_TopSecretbuilt-in variable, 145
- start\_at
  - Document.insertPDF args, 25
- store\_maxsizeTools attribute, 115
- store\_shrink()Tools method, 113
- store\_sizeTools attribute, 115
- stream
  - Document args, 18
  - open args, 18
  - Page.insertImage args, 44
- stridePixmap attribute, 57
- suppress
  - annotation, 176
- SVG
  - vector image, 182
- table
  - extract, 185



text  
    reading order, 184  
    rectangle, extract, 183  
TEXT\_ALIGN\_CENTERbuilt-in variable, 140  
TEXT\_ALIGN\_JUSTIFYbuilt-in variable, 140  
TEXT\_ALIGN\_LEFTbuilt-in variable, 140  
TEXT\_ALIGN\_RIGHTbuilt-in variable, 140  
text\_color  
    Annot.update args, 106  
text\_colorWidget attribute, 111  
text\_fontWidget attribute, 111  
text\_fontsizeWidget attribute, 111  
text\_maxlenWidget attribute, 111  
TEXT\_PRESERVE\_IMAGESbuilt-in variable, 141  
TEXT\_PRESERVE\_LIGATURESbuilt-in variable, 140  
TEXT\_PRESERVE\_WHITESPACEbuilt-in variable, 141  
text\_typeWidget attribute, 111  
TextPagebuilt-in class, 132  
tintWith()Pixmap method, 55  
titleOutline attribute, 35  
tlIRect attribute, 76  
tlRect attribute, 81  
to\_page  
    Document.convertToPDF args, 19  
    Document.insertPDF args, 25  
Toolsbuilt-in class, 113  
top\_leftIRect attribute, 76  
top\_leftRect attribute, 81  
top\_rightIRect attribute, 76  
top\_rightRect attribute, 81  
totalcontShape attribute, 99  
trIRect attribute, 76  
trRect attribute, 81  
transform()Point method, 85  
transform()Quad method, 88  
transform()Rect method, 80  
typeAnnot attribute, 107  
  
ufilename  
    Annot.fileUpd args, 107  
    Document.embeddedFileAdd args, 28  
    Document.embeddedFileUpd args, 29  
ulQuad attribute, 88  
unitPoint attribute, 85  
update()Annot method, 106  
updateLink()Page method, 42  
updateWidget()Annot method, 106  
urQuad attribute, 88  
uriLink attribute, 64  
urilinkDest attribute, 65  
uriOutline attribute, 36  
  
vector  
    image SVG, 182  
versionbuilt-in variable, 140  
VersionBindbuilt-in variable, 139  
VersionDatebuilt-in variable, 140  
VersionFitzbuilt-in variable, 139  
verticesAnnot attribute, 108  
  
wPixmap attribute, 57  
widgetAnnot attribute, 108  
Widgetbuilt-in class, 110  
widget\_choicesAnnot attribute, 108  
WIDGET\_Ff\_Combbuilt-in variable, 146  
WIDGET\_Ff\_Combobuilt-in variable, 147  
WIDGET\_Ff\_CommitOnSelChangebuilt-in variable, 147  
WIDGET\_Ff\_DoNotScrollbuilt-in variable, 146  
WIDGET\_Ff\_DoNotSpellCheckbuilt-in variable, 146  
WIDGET\_Ff\_Editbuilt-in variable, 147  
WIDGET\_Ff\_FileSelectbuilt-in variable, 146  
WIDGET\_Ff\_Multilinebuilt-in variable, 146  
WIDGET\_Ff\_MultiSelectbuilt-in variable, 147  
WIDGET\_Ff\_NoExportbuilt-in variable, 146  
WIDGET\_Ff\_NoToggleToOffbuilt-in variable, 147  
WIDGET\_Ff\_Passwordbuilt-in variable, 146  
WIDGET\_Ff\_Pushbuttonbuilt-in variable, 147  
WIDGET\_Ff\_Radiobuilt-in variable, 147  
WIDGET\_Ff\_RadioInUnisonbuilt-in variable, 147  
WIDGET\_Ff\_ReadOnlybuilt-in variable, 146  
WIDGET\_Ff\_Requiredbuilt-in variable, 146  
WIDGET\_Ff\_RichTextbuilt-in variable, 147  
WIDGET\_Ff\_Sortbuilt-in variable, 147  
widget\_nameAnnot attribute, 108  
widget\_typeAnnot attribute, 108  
widget\_valueAnnot attribute, 108  
width  
    Document.insertPage args, 26  
    Document.layout args, 23  
    Document.newPage args, 27  
    open args, 18  
    Page.drawBezier args, 43  
    Page.drawCircle args, 42  
    Page.drawCurve args, 43  
    Page.drawLine args, 42  
    Page.drawOval args, 42  
    Page.drawPolyline args, 42  
    Page.drawRect args, 43  
    Page.drawSector args, 42  
    Page.drawSquiggle args, 42  
    Page.drawZigzag args, 42  
    Shape.finish args, 96  
widthIRect attribute, 77  
widthPixmap attribute, 57  
widthQuad attribute, 89  
widthRect attribute, 82



- widthShape attribute, [99](#)
- write()Document method, [25](#)
- writeImage()Pixmap method, [56](#)
- writePNG()Pixmap method, [56](#)
- wrong
  - file extension, [197](#)
  
- xPixmap attribute, [58](#)
- xPoint attribute, [86](#)
- xOIRect attribute, [77](#)
- xORect attribute, [82](#)
- x1IRect attribute, [77](#)
- x1Rect attribute, [82](#)
- xrefAnnot attribute, [109](#)
- xrefLink attribute, [64](#)
- xrefPage attribute, [50](#)
- xresPixmap attribute, [58](#)
  
- yPixmap attribute, [58](#)
- yPoint attribute, [86](#)
- yOIRect attribute, [77](#)
- yORect attribute, [82](#)
- y1IRect attribute, [77](#)
- y1Rect attribute, [82](#)
- yresPixmap attribute, [58](#)
  
- zoom, [175](#)
  - resolution, [176](#)