



PyMuPDF

PyMuPDF Documentation

Release 1.21.1

Artifex

Jan 09, 2023

CONTENTS

1	Introduction	1
1.1	Note on the Name <i>fitz</i>	2
1.2	License and Copyright	2
1.3	Covered Version	2
2	Installation	3
2.1	Installation when a suitable wheel is not available	3
2.2	Notes	4
2.3	Install from source without using an sdist	4
2.4	Running tests	5
2.5	Enabling Integrated OCR Support	5
3	Tutorial	7
3.1	Importing the Bindings	7
3.2	Opening a Document	7
3.3	Some Document Methods and Attributes	8
3.4	Accessing Meta Data	8
3.5	Working with Outlines	9
3.6	Working with Pages	9
3.6.1	Inspecting the Links, Annotations or Form Fields of a Page	10
3.6.2	Rendering a Page	10
3.6.3	Saving the Page Image in a File	11
3.6.4	Displaying the Image in GUIs	11
3.6.4.1	wxPython	11
3.6.4.2	Tkinter	11
3.6.4.3	PyQt4, PyQt5, PySide	11
3.6.5	Extracting Text and Images	12
3.6.6	Searching for Text	12
3.7	Working with Stories	13
3.8	PDF Maintenance	14
3.8.1	Modifying, Creating, Re-arranging and Deleting Pages	14
3.8.2	Joining and Splitting PDF Documents	15
3.8.3	Embedding Data	15
3.8.4	Saving	15
3.9	Closing	16
3.10	Further Reading	16
4	Recipes	17
4.1	Recipes: Images	17
4.1.1	How to Make Images from Document Pages	17

4.1.2	How to Increase Image Resolution	17
4.1.3	How to Create Partial Pixmaps (Clips)	18
4.1.4	How to Zoom a Clip to a GUI Window	19
4.1.5	How to Create or Suppress Annotation Images	19
4.1.6	How to Extract Images: Non-PDF Documents	20
4.1.7	How to Extract Images: PDF Documents	20
4.1.8	How to Handle Image Masks	22
4.1.9	How to Make one PDF of all your Pictures (or Files)	22
4.1.10	How to Create Vector Images	25
4.1.11	How to Convert Images	26
4.1.12	How to Use Pixmaps: Glueing Images	27
4.1.13	How to Use Pixmaps: Making a Fractal	28
4.1.14	How to Interface with NumPy	30
4.1.15	How to Add Images to a PDF Page	30
4.2	Recipes: Text	31
4.2.1	How to Extract all Document Text	31
4.2.2	How to Extract Text from within a Rectangle	32
4.2.3	How to Extract Text in Natural Reading Order	32
4.2.4	How to Extract Tables from Documents	33
4.2.5	How to Mark Extracted Text	33
4.2.6	How to Mark Searched Text	34
4.2.7	How to Mark Non-horizontal Text	35
4.2.8	How to Analyze Font Characteristics	36
4.2.9	How to Insert Text	37
4.2.9.1	How to Write Text Lines	38
4.2.9.2	How to Fill a Text Box	40
4.2.9.3	How to Use Non-Standard Encoding	41
4.3	Recipes: Annotations	42
4.3.1	How to Add and Modify Annotations	42
4.3.2	How to Use FreeText	47
4.3.3	Using Buttons and JavaScript	48
4.3.4	How to Use Ink Annotations	50
4.4	Recipes: Drawing and Graphics	51
4.4.1	Extracting Drawings	53
4.5	Recipes: Multiprocessing	56
4.6	Recipes: General	61
4.6.1	How to Open with a Wrong File Extension	61
4.6.2	How to Embed or Attach Files	62
4.6.3	How to Delete and Re-Arrange Pages	62
4.6.4	How to Join PDFs	63
4.6.5	How to Add Pages	64
4.6.6	How To Dynamically Clean Up Corrupt PDFs	65
4.6.7	How to Split Single Pages	66
4.6.8	How to Combine Single Pages	67
4.6.9	How to Convert Any Document to PDF	69
4.6.10	How to Deal with Messages Issued by MuPDF	70
4.6.11	How to Deal with PDF Encryption	71
4.7	Recipes: Common Issues and their Solutions	73
4.7.1	Changing Annotations: Unexpected Behaviour	73
4.7.1.1	Problem	73
4.7.1.2	Cause	73
4.7.1.3	Solutions	73
4.7.2	Misplaced Item Insertions on PDF Pages	74
4.7.2.1	Problem	74

4.7.2.2	Cause	74
4.7.2.3	Solutions	75
4.7.3	Missing or Unreadable Extracted Text	75
4.7.3.1	Problem: no text is extracted	76
4.7.3.2	Cause	76
4.7.3.3	Solution	76
4.7.3.4	Problem: unreadable text	76
4.7.3.5	Cause	76
4.7.3.6	Solution	76
4.8	Recipes: Low-Level Interfaces	76
4.8.1	How to Iterate through the <code>xref</code> Table	77
4.8.2	How to Handle Object Streams	78
4.8.3	How to Handle Page Contents	78
4.8.4	How to Access the PDF Catalog	79
4.8.5	How to Access the PDF File Trailer	79
4.8.6	How to Access XML Metadata	80
4.8.7	How to Extend PDF Metadata	81
4.8.8	How to Read and Update PDF Objects	82
4.9	Recipes: Journalling	85
4.9.1	Example Session 1	86
4.9.2	Example Session 2	88
4.10	Recipes: Stories	89
4.10.1	How to add a line of text with some formatting	89
4.10.2	How to use images	91
4.10.3	Reading external HTML and CSS for a Story	92
4.10.4	How to output database content with Story templates	92
4.10.5	How to integrate with existing PDFs	95
4.10.6	How to make multi-columned layouts and access fonts from package <code>pymupdf-fonts</code>	97
4.10.7	How make a layout which wraps around a predefined “no go area” layout	99
4.10.8	How to output a table	103
4.10.9	How to create a simple grid layout	105
4.10.10	How to generate a Table of Contents	106
4.10.11	How to display a list from JSON data	111
4.10.12	Using the alternative <code>Story.write*()</code> functions	115
4.10.12.1	How to do basic layout with <code>Story.write()</code>	115
4.10.12.2	How to do iterative layout for a table of contents with <code>Story.write_stabilized()</code>	117
4.10.12.3	How to do iterative layout and create PDF links with <code>Story.write_stabilized_links()</code>	119
5	Module <i>fitz</i>	123
5.1	Invocation	123
5.2	Cleaning and Copying	124
5.3	Extracting Fonts and Images	125
5.4	Joining PDF Documents	125
5.5	Low Level Information	126
5.6	Embedded Files Commands	127
5.6.1	Information	128
5.6.2	Extraction	129
5.6.3	Deletion	129
5.6.4	Insertion	130
5.6.5	Updates	130
5.6.6	Copying	131
5.7	Text Extraction	131

6	Classes	135
6.1	Annot	135
6.1.1	Annotation Icons in MuPDF	146
6.1.2	Example	147
6.2	Archive	148
6.3	Colorspace	150
6.4	DisplayList	151
6.5	Document	152
6.5.1	set_metadata() Example	198
6.5.2	set_toc() Demonstration	198
6.5.3	insert_pdf() Examples	199
6.5.4	Other Examples	199
6.6	DocumentWriter	200
6.7	Font	201
6.8	Identity	208
6.9	IRect	209
6.10	Link	213
6.11	linkDest	215
6.12	Matrix	217
6.12.1	Examples	221
6.12.2	Shifting	221
6.12.3	Flipping	222
6.12.4	Shearing	223
6.12.5	Rotating	224
6.13	Outline	225
6.14	Page	226
6.14.1	Modifying Pages	226
6.14.2	Description of <i>get_links()</i> Entries	262
6.14.3	Notes on Supporting Links	263
6.14.3.1	Reading (pertains to method <i>get_links()</i> and the <i>first_link</i> property chain)	263
6.14.3.2	Writing	263
6.14.4	Homologous Methods of Document and Page	264
6.15	Pixmap	264
6.15.1	Supported Input Image Formats	277
6.15.2	Supported Output Image Formats	278
6.16	Point	278
6.17	Quad	281
6.17.1	Remark	284
6.17.2	Algebra and Containment Checks	284
6.18	Rect	284
6.19	Shape	291
6.19.1	Usage	304
6.19.2	Examples	304
6.19.3	Common Parameters	306
6.20	Story	309
6.20.1	Element Positioning CallBack function	313
6.20.1.1	Attributes of the ElementPosition class	313
6.21	TextPage	314
6.21.1	Structure of Dictionary Outputs	317
6.21.1.1	Page Dictionary	318
6.21.1.2	Block Dictionaries	319
6.21.1.3	Line Dictionary	320
6.21.1.4	Span Dictionary	321
6.21.1.5	Character Dictionary for <i>extractRAW_DICT()</i>	322

6.22	TextWriter	323
6.23	Tools	328
6.23.1	Example Session	333
6.24	Widget	334
6.24.1	Standard Fonts for Widgets	337
6.24.2	Supported Widget Types	337
6.25	Xml	338
6.25.1	Setting Text properties	344
6.25.2	Context Manager support	345
7	Operator Algebra for Geometry Objects	347
7.1	General Remarks	347
7.2	Unary Operations	347
7.3	Binary Operations	348
7.4	Some Examples	349
7.4.1	Manipulation with numbers	349
7.4.2	Manipulation with “like” Objects	349
8	Low Level Functions and Classes	353
8.1	Functions	353
8.2	Device	372
8.3	Working together: DisplayList and TextPage	373
8.3.1	Create a DisplayList	373
8.3.2	Generate Pixmap	373
8.3.3	Perform Text Search	373
8.3.4	Extract Text	374
8.3.5	Further Performance improvements	374
8.3.5.1	Pixmap	374
8.3.5.2	TextPage	374
9	Glossary	375
10	Constants and Enumerations	379
10.1	Constants	379
10.2	Document Permissions	380
10.3	PDF encryption method codes	380
10.4	Font File Extensions	381
10.5	Text Alignment	381
10.6	Text Extraction Flags	381
10.7	Link Destination Kinds	382
10.8	Link Destination Flags	383
10.9	Annotation Related Constants	384
10.9.1	Annotation Types	384
10.9.2	Annotation Flag Bits	385
10.9.3	Annotation Line Ending Styles	385
10.10	Widget Constants	385
10.10.1	Widget Types (<i>field_type</i>)	385
10.10.2	Text Widget Subtypes (<i>text_format</i>)	386
10.10.3	Widget flags (<i>field_flags</i>)	386
10.11	PDF Standard Blend Modes	386
10.12	Stamp Annotation Icons	387
11	Color Database	389
11.1	Function <i>getColor()</i>	389
11.2	Printing the Color Database	390

12 Appendix 1: Details on Text Extraction	391
12.1 General structure of a TextPage	391
12.2 Plain Text	391
12.3 BLOCKS	392
12.4 WORDS	392
12.5 HTML	393
12.6 Controlling Quality of HTML Output	393
12.7 DICT (or JSON)	394
12.8 RAWDICT (or RAWJSON)	395
12.9 XML	395
12.10 XHTML	396
12.11 Text Extraction Flags Defaults	396
12.12 Performance	397
13 Appendix 2: Considerations on Embedded Files	399
13.1 General	399
13.2 MuPDF Support	399
13.3 PyMuPDF Support	400
14 Appendix 3: Assorted Technical Information	401
14.1 Image Transformation Matrix	401
14.2 PDF Base 14 Fonts	402
14.3 Adobe PDF References	403
14.4 Using Python Sequences as Arguments in PyMuPDF	403
14.5 Ensuring Consistency of Important Objects in PyMuPDF	404
14.6 Design of Method Page.show_pdf_page()	406
14.6.1 Purpose and Capabilities	406
14.6.2 Technical Implementation	406
14.7 Redirecting Error and Warning Messages	407
15 Change Log	409
16 Deprecated Names	445
Index	453

INTRODUCTION



PyMuPDF is a Python binding for [MuPDF](#) – a lightweight PDF, XPS, and E-book viewer, renderer, and toolkit, which is maintained and developed by Artifex Software, Inc

MuPDF can access files in PDF, XPS, OpenXPS, CBZ, EPUB, MOBI and FB2 (e-books) formats, and it is known for its top performance and high rendering quality.

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

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

With PyMuPDF you can access files with extensions like “.pdf”, “.xps”, “.oxps”, “.cbz”, “.fb2”, “.mobi” or “.epub”. In addition, about 10 popular image formats can also be opened and handled like documents.

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

PyMuPDF runs and has been tested on Mac, Linux and Windows for Python versions 3.7¹ and up. Other platforms should work too, as long as MuPDF and Python support them.

PyMuPDF is hosted on [GitHub](#) and registered on [PyPI](#).

For MS Windows, Mac OSX and Linux Python wheels are available – please see the installation chapter.

The GitHub repository [PyMuPDF-Utilities](#) contains a full range of examples, demonstrations and use cases.

¹ PyMuPDF generally only supports Python versions that are still maintained by the Python Software Foundation. Once a Python version is being retired, PyMuPDF support will also be ended. This means that wheels for a retired Python platform will no longer be provided, and that Python language features may be used that did not exist in the retired Python version.

1.1 Note on the Name *fitz*

The top level Python import name for this library is “**fitz**”. This has historical reasons:

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](#)).

So PyMuPDF **cannot coexist** with packages named “fitz” in the same Python environment.

1.2 License and Copyright

In order to comply with MuPDF’s dual licensing model, PyMuPDF has entered into an agreement with Artifex who has the right to sublicense PyMuPDF to third parties.

PyMuPDF and MuPDF are now available under both, open-source AGPL and commercial license agreements. Please read the full text of the AGPL license agreement, available in the distribution material (file COPYING) and [here](#), to ensure that your use case complies with the guidelines of the license. If you determine you cannot meet the requirements of the AGPL, please contact [Artifex](#) for more information regarding a commercial license.

Artifex is the exclusive commercial licensing agent for MuPDF.

Artifex, the Artifex logo, MuPDF, and the MuPDF logo are registered trademarks of Artifex Software Inc. © 2022 Artifex Software, Inc. All rights reserved.

1.3 Covered Version

This documentation covers PyMuPDF v1.21.1 features as of **2022-12-13 00:00:01**.

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

INSTALLATION

PyMuPDF should be installed using pip with:

```
python -m pip install --upgrade pip
python -m pip install --upgrade pymupdf
```

This will install from a Python wheel if one is available for your platform.

2.1 Installation when a suitable wheel is not available

If a suitable Python wheel is not available, pip will automatically build from source using a Python sdist.

This requires C/C++ development tools and SWIG to be installed:

- On Unix-style systems such as Linux, OpenBSD and FreeBSD, use the system package manager to install SWIG.
 - For example on Debian Linux, do: `sudo apt install swig`
- On Windows:
 - Install Visual Studio 2019. If not installed in a standard location, set environmental variable `PYMUPDF_SETUP_DEVENV` to the location of the `devenv.com` binary.
 - Install SWIG by following the instructions at: https://swig.org/Doc4.0/Windows.html#Windows_installation
- On MacOS, install MacPorts using the instructions at: <https://www.macports.org/install.php>
 - Then install SWIG with: `sudo port install swig`
 - You may also need: `sudo port install swig-python`

As of PyMuPDF-1.20.0, the required MuPDF source code is already in the sdist and is automatically built into PyMuPDF.

2.2 Notes

Wheels are available for Windows (32-bit Intel, 64-bit Intel), Linux (64-bit Intel, 64-bit ARM) and Mac OSX (64-bit Intel, 64-bit ARM), Python versions 3.7 and up.

Wheels are not available for Python installed with [Chocolatey](#) on Windows. Instead install Python using the Windows installer from the python.org website, see: <http://www.python.org/downloads>

PyMuPDF does not support Python versions prior to 3.7. Older wheels can be found in [this](#) repository and on [PyPI](#). Please note that we generally follow the official Python release schedules. For Python versions dropping out of official support this means, that generation of wheels will also be ceased for them.

There are no **mandatory** external dependencies. However, some optional feature are available only if additional components are installed:

- [Pillow](#) is required for [Pixmap.pil_save\(\)](#) and [Pixmap.pil_tobytes\(\)](#).
- [fontTools](#) is required for [Document.subset_fonts\(\)](#).
- [pymupdf-fonts](#) is a collection of nice fonts to be used for text output methods.
- [Tesseract-OCR](#) for optical character recognition in images and document pages. Tesseract is separate software, not a Python package. To enable OCR functions in PyMuPDF, the software must be installed and the system environment variable "TESSDATA_PREFIX" must be defined and contain the tessdata folder name of the Tesseract installation location. See below.

Note: You can install these additional components at any time – before or after installing PyMuPDF. PyMuPDF will detect their presence during import or when the respective functions are being used.

2.3 Install from source without using an sdist

- First get a PyMuPDF source tree:
 - Clone the git repository at <https://github.com/pymupdf/PyMuPDF>, for example:

```
git clone https://github.com/pymupdf/PyMuPDF.git
```
 - Or download and extract a `.zip` or `.tar.gz` source release from <https://github.com/pymupdf/PyMuPDF/releases>.
- Install C/C++ development tools and SWIG as described above.
- Build and install PyMuPDF:

```
cd PyMuPDF && python setup.py install
```

This will automatically download a specific hard-coded MuPDF source release, and build it into PyMuPDF.

One can build with a non-default MuPDF (for example one installed on the system, or a local checkout) by setting environmental variables. See the comments at the start of `PyMuPDF/setup.py` for more information.

Note: When running Python scripts that use PyMuPDF, make sure that the current directory is not the PyMuPDF/ directory.

Otherwise, confusingly, Python will attempt to import `fitz` from the local `fitz/` directory, which will fail because it only contains source files.

2.4 Running tests

PyMuPDF has a set of `pytest` scripts within the `tests/` directory.

Run tests with:

```
pip install pytest fontTools
pytest PyMuPDF/tests
```

If PyMuPDF has been built with a non-default build of MuPDF (using environmental variable `PYMUPDF_SETUP_MUPDF_BUILD`), it is possible that `tests/test_textbox.py:test_textbox3()` will fail, because it relies on MuPDF having been built with PyMuPDF's customized configuration, `fitz/_config.h`.

One can skip this particular test by adding `-k 'not test_textbox3'` to the `pytest` command line.

2.5 Enabling Integrated OCR Support

If you do not intend to use this feature, skip this step. Otherwise, it is required for both installation paths: **from wheels and from sources**.

PyMuPDF will already contain all the logic to support OCR functions. But it additionally does need Tesseract's language support data, so installation of Tesseract-OCR is still required.

The language support folder location must currently¹ be communicated via storing it in the environment variable `TESSDATA_PREFIX`.

So for a working OCR functionality, make sure to complete this checklist:

1. Install Tesseract.
2. **Locate Tesseract's language support folder. Typically you will find it here:**
 - Windows: `C:\Program Files\Tesseract-OCR\tessdata`
 - Unix systems: `/usr/share/tesseract-ocr/4.00/tessdata`
3. **Set the environment variable `TESSDATA_PREFIX`**
 - Windows: `set TESSDATA_PREFIX=C:\Program Files\Tesseract-OCR\tessdata`
 - Unix systems: `export TESSDATA_PREFIX=/usr/share/tesseract-ocr/4.00/tessdata`

Note: This must happen outside Python – before starting your script. Just manipulating `os.environ` will not work!

¹ In the next MuPDF version, it will be possible to pass this value as a parameter – directly in the OCR invocations.

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

3.1 Importing the Bindings

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

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

3.2 Opening a Document

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

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

This creates the *Document* object *doc*. *filename* must be a Python string (or a `pathlib.Path`) 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. You can also use *Document* as a *context manager*.

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

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

3.3 Some Document Methods and Attributes

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

3.4 Accessing Meta Data

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

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

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

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

3.5 Working with Outlines

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

```
toc = doc.get_toc()
```

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 `toc import` (PDF only) resp. `toc export` table of contents from resp. to CSV files.

3.6 Working with Pages

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

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

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

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

Any integer $-\infty < \text{pno} < \text{page_count}$ is possible here. Negative numbers count backwards from the end, so `doc[-1]` is the last page, like with Python sequences.

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

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

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

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

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

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

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

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

links is a Python list of dictionaries. For details see [Page.get_links\(\)](#).

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

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

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

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

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

3.6.2 Rendering a Page

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

```
pix = page.get_pixmap()
```

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

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

Note: You can also create a **vector** image of a page by using [Page.get_svg_image\(\)](#). Refer to this [Vector Image Support page](#) for details.

3.6.3 Saving the Page Image in a File

We can simply store the image in a PNG file:

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

3.6.4 Displaying the Image in GUIs

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

3.6.4.1 wxPython

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

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

3.6.4.2 Tkinter

Please also see section 3.19 of the *Pillow documentation*:

```
from PIL import Image, ImageTk

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

The following avoids using *Pillow*:

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

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

3.6.4.3 PyQt4, PyQt5, PySide

Please also see section 3.16 of the *Pillow documentation*:

```
from PIL import Image, ImageQt

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

Again, you also can get along **without using Pillow**. Qt's *QImage* luckily supports native Python pointers, so the following is the recommended way to create Qt images:

```
from PyQt5.QtGui import QImage

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

3.6.5 Extracting Text and Images

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

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

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

- **“text”**: (default) plain text with line breaks. No formatting, no text position details, no images.
- **“blocks”**: generate a list of text blocks (= paragraphs).
- **“words”**: generate a list of words (strings not containing spaces).
- **“html”**: creates a full visual version of the page including any images. This can be displayed with your internet browser.
- **“dict”** / **“json”**: same information level as HTML, but provided as a Python dictionary or resp. JSON string. See [TextPage.extractDICT\(\)](#) for details of its structure.
- **“rawdict”** / **“rawjson”**: a super-set of **“dict”** / **“json”**. 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: Considerations on Embedded Files](#).

3.6.6 Searching for Text

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

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

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

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

² [Page.get_text\(\)](#) 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.get_text\(\)](#): [Page.get_text\("dict"\)](#) is equivalent to [TextPage.extractDICT\(\)](#).

3.7 Working with Stories

The *Story* class is a new feature of PyMuPDF version 1.21.0. It represents support for MuPDF’s “story” interface.

The following is a quote from the book “MuPDF Explored” by Robin Watts from [Artifex](#):

Stories provide a way to easily layout styled content for use with devices, such as those offered by Document Writers (...). The concept of a story comes from desktop publishing, which in turn (...) gets it from newspapers. If you consider a traditional newspaper layout, it will consist of various news articles (stories) that are laid out into multiple columns, possibly across multiple pages.

Accordingly, MuPDF uses a story to represent a flow of text with styling information. The user of the story can then supply a sequence of rectangles into which the story will be laid out, and the positioned text can then be drawn to an output device. This keeps the concept of the text itself (the story) to be separated from the areas into which the text should be flowed (the layout).

Note: A Story works somewhat similar to an internet browser: It faithfully parses and renders HTML hypertext and also optional stylesheets (CSS). But its **output is a PDF** – not web pages.

When creating a *Story*, the input from up to three different information sources is taken into account. All these items are optional.

1. HTML source code, provided as a Python string, from which a so-called **Document Object Model (DOM)** is created. As usual, this string may be read from a file, be stored in a Python variable of the script, **or** be programmatically created by the script itself via an API (*Xml*).
2. CSS (Cascaded Style Sheet) source code, provided as a Python string. CSS can be used to provide styling information (text font size, color, etc.) like it would happen for web pages. Obviously, this string may also be read from a file.
3. An *Archive* **must be used** whenever the DOM references images, or uses text fonts except the standard *PDF Base 14 Fonts*, CJK fonts and the NOTO fonts generated into the PyMuPDF binary.

The *API* allows creating DOMs completely from scratch, including desired styling information. It can also be used to modify or extend **provided** HTML: text can be deleted or replaced, or its styling can be changed. Text – for example extracted from databases – can also be added and fill template-like HTML documents.

After the story DOM is considered complete, it can be used to create a PDF document. This happens via the new *DocumentWriter* class. During the output page creation, the programmer will provide a number of rectangles where the story should place its content.

The story in turn will return completion codes indicating whether or not more content is waiting to be written. Which part of the content will land in which rectangle or on which page is automatically determined by the story itself – it cannot be influenced other than by providing the rectangles.

Please see the *Stories recipes* for a number of typical use cases.

3.8 PDF Maintenance

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

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

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

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

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

3.8.1 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.delete_page\(\)`](#) and [`Document.delete_pages\(\)`](#) delete pages.

[`Document.copy_page\(\)`](#), [`Document.fullcopy_page\(\)`](#) and [`Document.move_page\(\)`](#) copy or move a page to other locations within the same document.

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

So you can easily create new PDFs with

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

... whatever you can think of.

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

[`Document.insert_page\(\)`](#) and [`Document.new_page\(\)`](#) 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 “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.

3.8.2 Joining and Splitting PDF Documents

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

```
# append complete doc2 to the end of doc1
doc1.insert_pdf(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.insert_pdf(doc1, to_page = 9) # first 10 pages
doc2.insert_pdf(doc1, from_page = len(doc1) - 10) # last 10 pages
doc2.save("first-and-last-10.pdf")
```

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

3.8.3 Embedding Data

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

PyMuPDF fully supports this feature via *Document* `embfile_*` methods and attributes. For some detail read Appendix 3, consult the Wiki on [dealing with embedding files](#), or the example scripts `embedded-copy.py`, `embedded-export.py`, `embedded-import.py`, and `embedded-list.py`.

3.8.4 Saving

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

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

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

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

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

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

3.9 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.

3.10 Further Reading

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

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

4.1 Recipes: Images

4.1.1 How to Make Images from Document Pages

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

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

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

```
import sys, fitz # import the bindings
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.get_pixmap() # render page to an image
    pix.save("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 with width and height rounded to integers, e.g. 595 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.

4.1.2 How to Increase Image Resolution

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

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

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

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

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

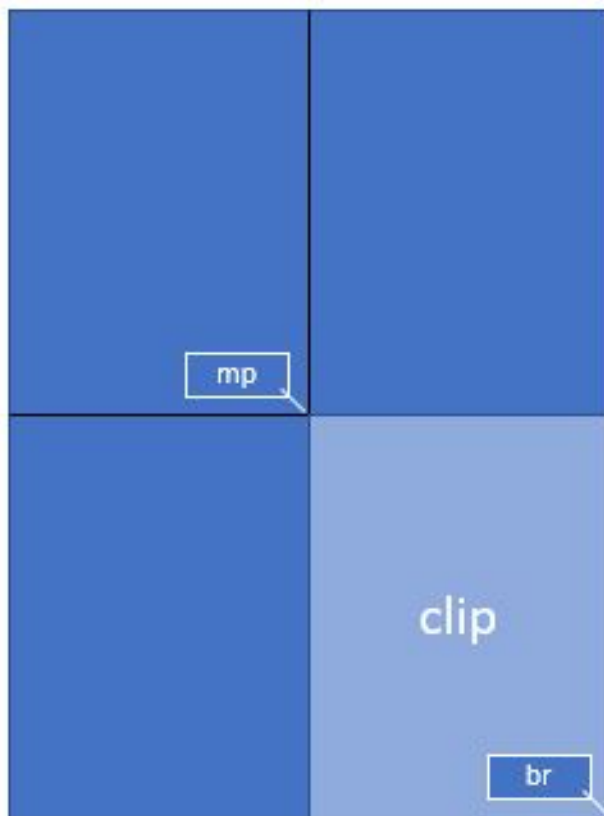
Since version 1.19.2 there is a more direct way to set the resolution: Parameter "dpi" (dots per inch) can be used in place of "matrix". To create a 300 dpi image of a page specify `pix = page.get_pixmap(dpi=300)`. Apart from notation brevity, this approach has the additional advantage that the **dpi value is saved with the image file** – which does not happen automatically when using the Matrix notation.

4.1.3 How to Create Partial Pixmaps (Clips)

You do not always need or want the full image of a page. This is the case e.g. when you display the image in a GUI and would like to fill the respective window with a zoomed part of the page.

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

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



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

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

4.1.4 How to Zoom a Clip to a GUI Window

Please also read the previous section. This time we want to **compute the zoom factor** for a clip, such that its image best fits a given GUI window. This means, that the image's width or height (or both) will equal the window dimension. For the following code snippet you need to provide the WIDTH and HEIGHT of your GUI's window that should receive the page's clip rectangle.

```
# WIDTH: width of the GUI window
# HEIGHT: height of the GUI window
# clip: a subrectangle of the document page
# compare width/height ratios of image and window

if clip.width / clip.height < WIDTH / HEIGHT:
    # clip is narrower: zoom to window HEIGHT
    zoom = HEIGHT / clip.height
else: # clip is broader: zoom to window WIDTH
    zoom = WIDTH / clip.width
mat = fitz.Matrix(zoom, zoom)
pix = page.get_pixmap(matrix=mat, clip=clip)
```

For the other way round, now assume you **have** the zoom factor and need to **compute the fitting clip**.

In this case we have $\text{zoom} = \text{HEIGHT} / \text{clip.height} = \text{WIDTH} / \text{clip.width}$, so we must set $\text{clip.height} = \text{HEIGHT} / \text{zoom}$ and, $\text{clip.width} = \text{WIDTH} / \text{zoom}$. Choose the top-left point `tl` of the clip on the page to compute the right pixmap:

```
width = WIDTH / zoom
height = HEIGHT / zoom
clip = fitz.Rect(tl, tl.x + width, tl.y + height)
# ensure we still are inside the page
clip &= page.rect
mat = fitz.Matrix(zoom, zoom)
pix = fitz.Pixmap(matrix=mat, clip=clip)
```

4.1.5 How to Create or Suppress Annotation Images

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

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

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

4.1.6 How to Extract Images: Non-PDF Documents

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

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

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

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

2. Use `Page.get_text()` with the “dict” parameter. This works for all document types. It 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.get_text("dict")
>>> blocks = d["blocks"] # the list of block dictionaries
>>> imgblocks = [b for b in blocks if b["type"] == 1]
>>> pprint(imgblocks[0])
{'bbox': (100.0, 135.8769989013672, 300.0, 364.1230163574219),
 'bpc': 8,
 'colorspace': 3,
 'ext': 'jpeg',
 'height': 501,
 'image': b'\xff\xd8\xff\xe0\x00\x10JFIF...', # CAUTION: LARGE!
 'size': 80518,
 'transform': (200.0, 0.0, -0.0, 228.2460174560547, 100.0, 135.8769989013672),
 'type': 1,
 'width': 439,
 'xres': 96,
 'yres': 96}
```

4.1.7 How to Extract Images: PDF Documents

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

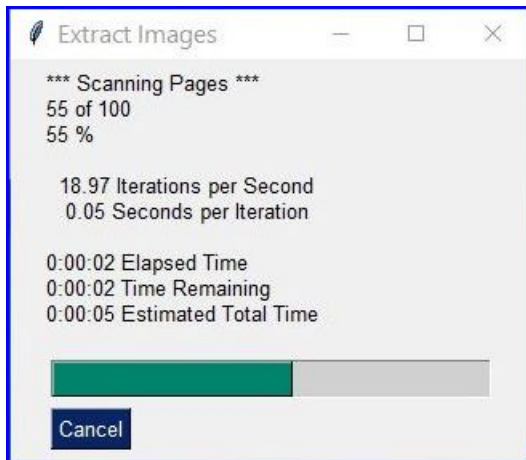
1. **Create** a *Pixmap* of the image with instruction `pix = fitz.Pixmap(doc, xref)`. This method is **very** fast (single digit micro-seconds). The pixmap’s properties (width, height, ...) will reflect the ones of the image. In this case there is no way to tell which image format the embedded original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

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

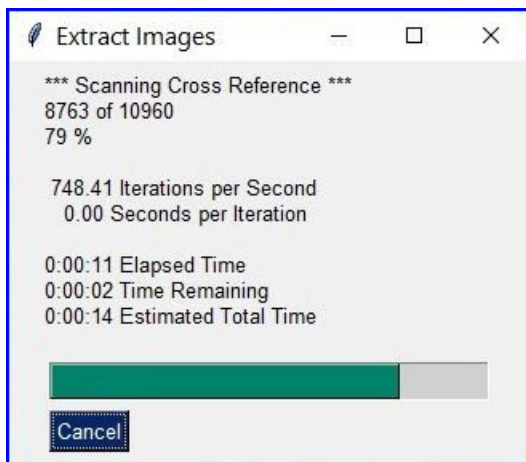
- a. “**Inspect the page objects:**” Loop through the items of `Page.get_images()`. It is a list of list, and its items look like `[xref, smask, ...]`, containing the `xref` of an image. This `xref` can then be used with one of the above methods. Use this method for **valid (undamaged)** documents. Be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

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

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



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



4.1.8 How to Handle Image Masks

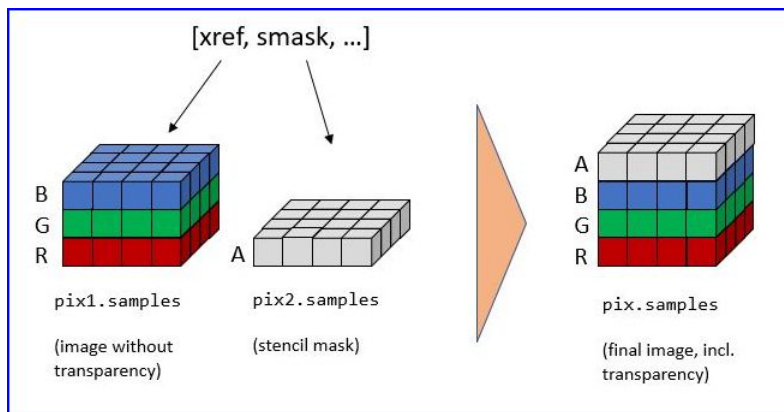
Some images in PDFs are accompanied by **image masks**. In their simplest form, masks represent alpha (transparency) bytes stored as separate images. In order to reconstruct the original of an image, which has a mask, it must be “enriched” with transparency bytes taken from its mask.

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

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

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

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



```
>>> pix1 = fitz.Pixmap(doc.extract_image(xref)["image"]) # (1) pixmap of image w/o
↳ alpha
>>> mask = fitz.Pixmap(doc.extract_image(smask)["image"]) # (2) mask pixmap
>>> pix = fitz.Pixmap(pix1, mask) # (3) copy of pix1, image
↳ mask added
```

Step (1) creates a pixmap of the basic image. Step (2) does the same with the image mask. Step (3) adds an alpha channel and fills it with transparency information.

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

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

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

Method 1: Inserting Images as Pages

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

```
import os, fitz
import PySimpleGUI as psg # for showing a progress bar
doc = fitz.open() # PDF with the pictures
```

(continues on next page)

(continued from previous page)

```

imgdir = "D:/2012_10_05" # where the pics are
imglist = os.listdir(imgdir) # list of them
imgcount = len(imglist) # pic count

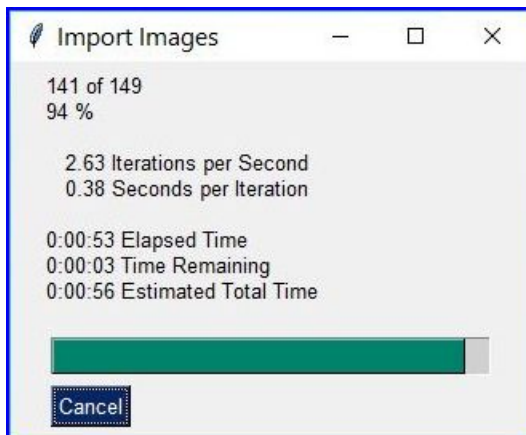
for i, f in enumerate(imglist):
    img = fitz.open(os.path.join(imgdir, f)) # open pic as document
    rect = img[0].rect # pic dimension
    pdfbytes = img.convert_to_pdf() # make a PDF stream
    img.close() # no longer needed
    imgPDF = fitz.open("pdf", pdfbytes) # open stream as PDF
    page = doc.new_page(width = rect.width, # new page with ...
                        height = rect.height) # pic dimension
    page.show_pdf_page(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](#) for a more complete source code: it offers a directory selection dialog and skips unsupported files and non-file entries.

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

Method 2: Embedding Files

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

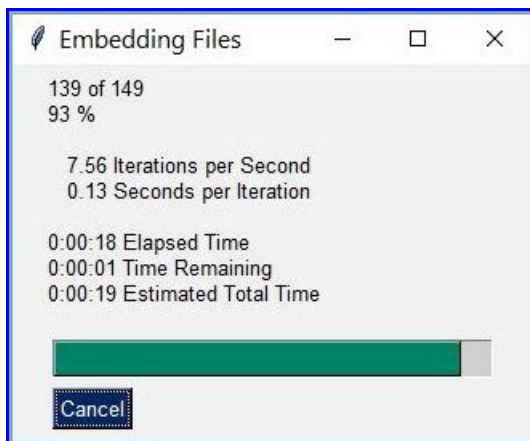
```
import os, fitz
import PySimpleGUI as psq # for showing progress bar
doc = fitz.open() # PDF with the pictures
imgdir = "D:/2012_10_05" # where my files are

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

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

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

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

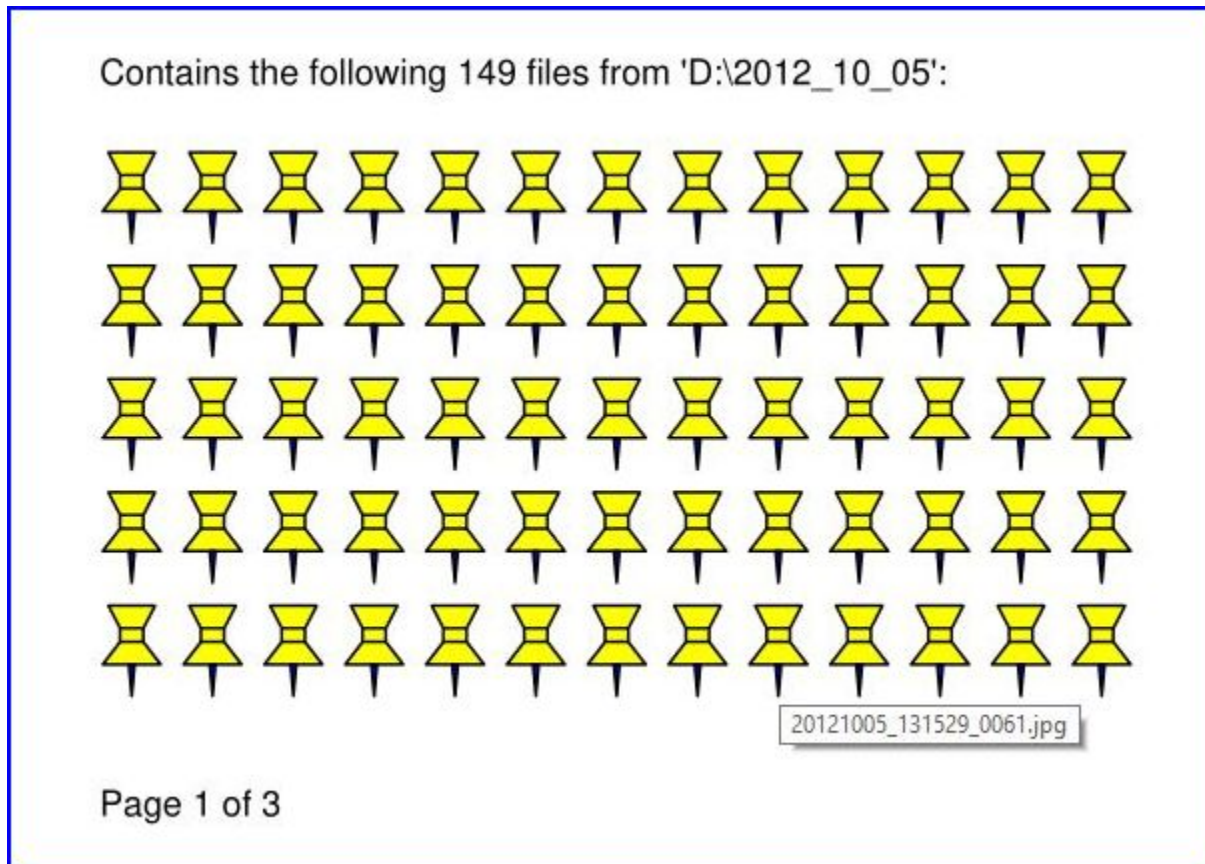


This is by far the fastest method, and it also produces the smallest possible output file size. The above pictures needed 20 seconds on my machine and yielded a PDF size of 510 MB. Look [here](#) for a more complete source code: it offers a directory selection dialog and skips non-file entries.

Method 3: Attaching Files

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

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



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

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

4.1.10 How to Create Vector Images

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

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

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

4.1.11 How to Convert Images

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

Notwithstanding that interfacing with Pillow is almost trivial.

Input Formats	Output Formats	Description
BMP	.	Windows Bitmap
JPEG	.	Joint Photographic Experts Group
JXR	.	JPEG Extended Range
JPX/JP2	.	JPEG 2000
GIF	.	Graphics Interchange Format
TIFF	.	Tagged Image File Format
PNG	PNG	Portable Network Graphics
PNM	PNM	Portable Anymap
PGM	PGM	Portable Graymap
PBM	PBM	Portable Bitmap
PPM	PPM	Portable Pixmap
PAM	PAM	Portable Arbitrary Map
.	PSD	Adobe Photoshop Document
.	PS	Adobe Postscript

The general scheme is just the following two lines:

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

Remarks

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

Note: Convert JPEG to Photoshop:

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

Note: Save to JPEG using PIL/Pillow:

```
pix = fitz.Pixmap(...)
pix.pil_save("output.jpg")
```

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

```
import tkinter as tk
pix = fitz.Pixmap("input.jpg") # or any RGB / no-alpha image
tkimg = tk.PhotoImage(data=pix.tobytes("ppm"))
```

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

```
import tkinter as tk
pix = fitz.Pixmap("input.png") # may have an alpha channel
if pix.alpha: # we have an alpha channel!
    pix = fitz.Pixmap(pix, 0) # remove it
tkimg = tk.PhotoImage(data=pix.tobytes("ppm"))
```

4.1.12 How to Use Pixmaps: Glueing Images

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

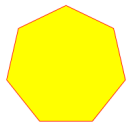
```
import fitz
src = fitz.Pixmap("img-7edges.png") # create pixmap from a picture
col = 3 # tiles per row
lin = 4 # tiles per column
tar_w = src.width * col # width of target
tar_h = src.height * lin # height of target

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

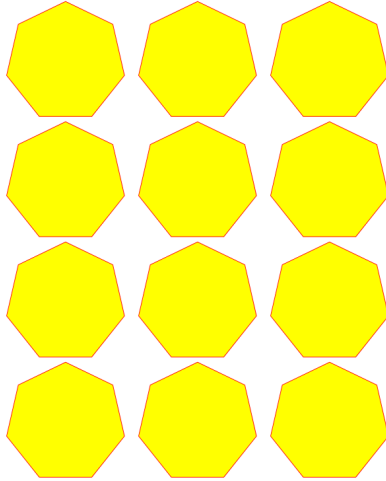
# now fill target with the tiles
for i in range(col):
    for j in range(lin):
        src.set_origin(src.width * i, src.height * j)
        tar_pix.copy(src, src.irect) # copy input to new loc

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

This is the input picture:



Here is the output:



4.1.13 How to Use Pixmaps: Making a Fractal

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

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

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

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

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

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

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

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

    Arguments are top-left coords and the step width.

    Some alternative punching methods are commented out.
```

(continues on next page)

(continued from previous page)

```

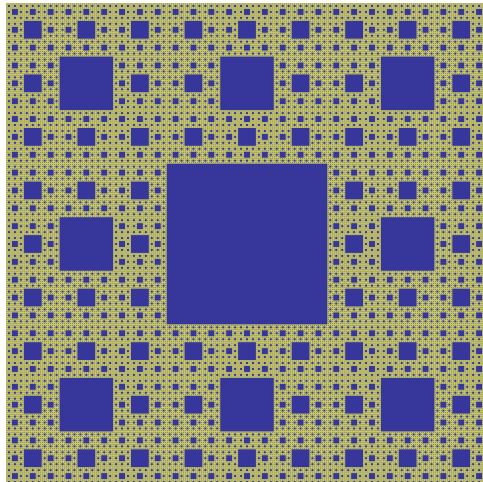
"""
s = step // 3          # the new step
# iterate through the 9 sub-squares
# the central one will be filled with the color
for i in range(3):
    for j in range(3):
        if i != j or i != 1: # this is not the central cube
            if s >= 3:        # recursing needed?
                punch(x+i*s,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

    return

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

```

The result should look something like this:



4.1.14 How to Interface with NumPy

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

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

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

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

4.1.15 How to Add Images to a PDF Page

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

Criterion	<code>Page.insert_image()</code>	<code>Page.show_pdf_page()</code>
displayable content	image file, image in memory, pixmap	PDF page
display resolution	image resolution	vectorized (except raster page content)
rotation	0, 90, 180 or 270 degrees	any angle
clipping	no (full image only)	yes
keep aspect ratio	yes (default option)	yes (default option)
transparency (water marking)	depends on the image	depends on the page
location / placement	scaled to fit target rectangle	scaled to fit target rectangle
performance	automatic prevention of duplicates;	automatic prevention of duplicates;
multi-page image support	no	yes
ease of use	simple, intuitive;	simple, intuitive; usable for all document types (including images!) after conversion to PDF via <code>Document.convert_to_pdf()</code>

Basic code pattern for `Page.insert_image()`. Exactly one of the parameters `filename` / `stream` / `pixmap` must be given, if not re-inserting an existing image:

```

page.insert_image(
    rect,                # where to place the image (rect-like)
    filename=None,       # image in a file
    stream=None,         # image in memory (bytes)
    pixmap=None,         # image from pixmap
    mask=None,           # specify alpha channel separately
    rotate=0,            # rotate (int, multiple of 90)
    xref=0,              # re-use existing image
    oc=0,                # control visibility via OCG / OCMD
    keep_proportion=True, # keep aspect ratio
    overlay=True,        # put in foreground
)

```

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

```

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

```

4.2 Recipes: Text

4.2.1 How to Extract all Document Text

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

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

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

```

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

```

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

You have many options to rectify this – see chapter *Appendix 2: Considerations on Embedded Files*. 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.get_text("blocks")`. Each item of this list contains position information for its text, which can be used to establish a convenient reading order.
3. Extract a list of single words via `Page.get_text("words")`. Its items are words with position information. Use it to determine text contained in a given rectangle – see next section.

See the following two sections for examples and further explanations.

4.2.2 How to Extract Text from within a Rectangle

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

4.2.3 How to Extract Text in Natural Reading Order

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

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

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

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

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

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

1. Use `sort` parameter of `Page.get_text()`. It will sort the output from top-left to bottom-right (ignored for XHTML, HTML and XML output).
2. Use the `fitz` module in CLI: `python -m fitz gettext ...`, which produces a text file where text has been re-arranged in layout-preserving mode. Many options are available to control the output.

You can also use the above mentioned [script](#) with your modifications.

4.2.4 How to Extract Tables from Documents

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

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

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

4.2.5 How to Mark Extracted Text

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

This method has advantages and drawbacks. Pros are:

- The search string can contain blanks and wrap across lines
- Upper or lower case characters are treated equal
- Word hyphenation at line ends is detected and resolved
- Return may also be a list of `Quad` objects to precisely locate text that is **not parallel** to either axis – using `Quad` output is also recommend, when page rotation is not zero

But you also have other options:

```
import sys
import fitz

def mark_word(page, text):
    """Underline each word that contains 'text'."""
    found = 0
    wlist = page.get_text("words") # 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.add_underline_annot(r) # underline
    return found

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

print("underlining words containing '%s' in document '%s'" % (text, 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
```

(continues on next page)

(continued from previous page)

```

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.get_text("words")` 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. 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)³. We also are registered on [PyPI](https://pypi.org/project/PyMuPDF/)⁴.

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:

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

4.2.6 How to Mark Searched 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

```

(continues on next page)

(continued from previous page)

```

page = doc[0]

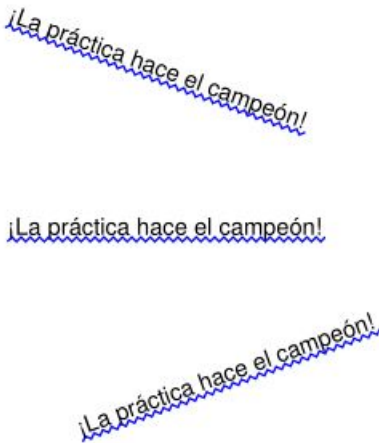
# get list of text locations
# we use "quads", not rectangles because text may be tilted!
rl = page.search_for(t, quads = True)

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

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

```

The result looks like this:



4.2.7 How to Mark Non-horizontal Text

The previous section already shows an example for marking non-horizontal text, that was detected by text **searching**.

But text **extraction** with the “dict” / “rawdict” options of `Page.get_text()` may also return text with a non-zero angle to the x-axis. This is indicated by the value of the line dictionary’s “dir” key: it is the tuple (cosine, sine) for that angle. If `line["dir"] != (1, 0)`, then the text of all its spans is rotated by (the same) angle $\neq 0$.

The “bboxes” returned by the method however are rectangles only – not quads. So, to mark span text correctly, its quad must be recovered from the data contained in the line and span dictionary. Do this with the following utility function (new in v1.18.9):

```

span_quad = fitz.recover_quad(line["dir"], span)
annot = page.add_highlight_annot(span_quad) # this will mark the complete span text

```

If you want to **mark the complete line** or a subset of its spans in one go, use the following snippet (works for v1.18.10 or later):

```

line_quad = fitz.recover_line_quad(line, spans=line["spans"][1:-1])
page.add_highlight_annot(line_quad)

```



The `spans` argument above may specify any sub-list of `line["spans"]`. In the example above, the second to second-to-last span are marked. If omitted, the complete line is taken.

4.2.8 How to Analyze Font Characteristics

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

```
import sys

import fitz

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

doc = fitz.open(sys.argv[1])
page = doc[0]

# read page text as a dictionary, suppressing extra spaces in CJK fonts
blocks = page.get_text("dict", flags=11)["blocks"]
for b in blocks: # iterate through the text blocks
```

(continues on next page)

(continued from previous page)

```

for l in b["lines"]: # iterate through the text lines
    for s in l["spans"]: # iterate through the text spans
        print("")
        font_properties = "Font: '%s' (%s), size %g, color #%06x" % (
            s["font"], # font name
            flags_decomposer(s["flags"]), # readable font flags
            s["size"], # font size
            s["color"], # font color
        )
        print("Text: '%s'" % s["text"]) # simple print of text
        print(font_properties)

```

Here is the PDF page and the script output:

```

Text using fontname 'cour'
Text using fontname 'coit'
Text using fontname 'cobo'
Text using fontname 'cobi'
Text using fontname 'tiro'
Text using fontname 'tiit'
Text using fontname 'tibo'
Text using fontname 'tibi'
Text using fontname 'helv'
Text using fontname 'heit'
Text using fontname 'hebo'
Text using fontname 'hebi'
* * * * *
Τεξτ χρησιμοποιώντας γραμματοσειρά
Text using fontname 'china-s': 我很喜欢德国! 德国是个好地方!
Text using fontname 'china-t': 我很喜欢德国! 德国是个好地方!
Text using fontname 'japan': 世紀末以降における熊野三山
Text using fontname 'korea': 애두름은 하나의 계정으로

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

4.2.9 How to Insert Text

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

- choose the font, including built-in fonts and fonts that are available as files
- choose text characteristics like bold, italic, font size, font color, etc.
- position the text in multiple ways:
 - either as simple line-oriented output starting at a certain point,
 - or fitting text in a box provided as a rectangle, in which case text alignment choices are also available,
 - choose whether text should be put in foreground (overlay existing content),

- all text can be arbitrarily “morphed”, i.e. its appearance can be changed via a *Matrix*, to achieve effects like scaling, shearing or mirroring,
- independently from morphing and in addition to that, text can be rotated by integer multiples of 90 degrees.

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

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

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

4.2.9.1 How to Write Text Lines

Output some text lines on a page:

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

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

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

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

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

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

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

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

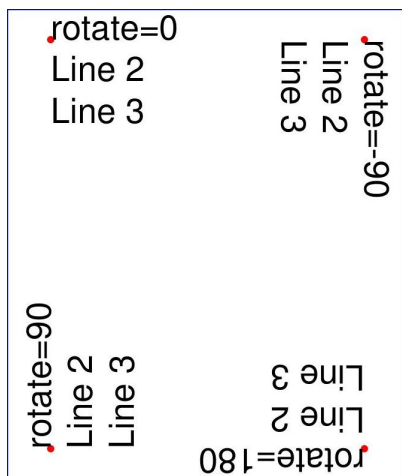
```
import fitz
doc = fitz.open()
page = doc.new_page()
# the text strings, each having 3 lines
text1 = "rotate=0\nLine 2\nLine 3"
text2 = "rotate=90\nLine 2\nLine 3"
text3 = "rotate=-90\nLine 2\nLine 3"
text4 = "rotate=180\nLine 2\nLine 3"
red = (1, 0, 0) # the color for the red dots
# the insertion 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
shape = page.new_shape()

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

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

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

This is the result:

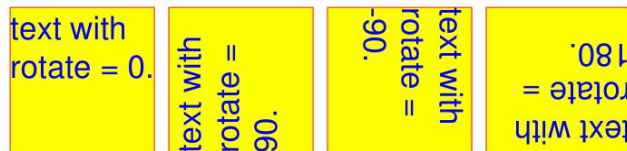


4.2.9.2 How to Fill a Text Box

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

```
import fitz
doc = fitz.open(...) # new or existing PDF
page = doc.new_page() # 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.
"""
shape = page.new_shape() # create Shape
shape.draw_rect(r1) # draw rectangles
shape.draw_rect(r2) # giving them
shape.draw_rect(r3) # a yellow background
shape.draw_rect(r4) # and a red border
shape.finish(width = 0.3, color = red, fill = gold)
# Now insert text in the rectangles. Font "Helvetica" will be used
# by default. A return code rc < 0 indicates insufficient space (not checked here).
rc = shape.insert_textbox(r1, t1, color = blue)
rc = shape.insert_textbox(r2, t2, color = blue, rotate = 90)
rc = shape.insert_textbox(r3, t3, color = blue, rotate = -90)
rc = shape.insert_textbox(r4, t4, color = blue, rotate = 180)
shape.commit() # write all stuff to page /Contents
doc.save(...)
```

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



4.2.9.3 How to Use Non-Standard Encoding

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

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

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

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

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

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

The result:

Sômé tèxt wìth nŏñ-Lâtîn characterß.

Στmı tθxt wμth nφρ-Lβtξn characterı.

STmИ tXxt wЛth nЖЯ-LБtHn characterЪ.

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

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

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

4.3 Recipes: Annotations

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

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

4.3.1 How to Add and Modify Annotations

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

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

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

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

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

Dependencies
-----
PyMuPDF v1.17.0
-----
```

(continues on next page)

(continued from previous page)

```

"""
from __future__ import print_function

import gc
import sys

import fitz

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

gc.set_debug(gc.DEBUG_UNCOLLECTABLE)

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

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

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

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

page.set_rotation(0)

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

r = r + displ
annot = page.add_freetext_annot(
    r,
    t1,
    fontsize=10,
    rotate=90,
    text_color=blue,
    fill_color=gold,
    align=fitz.TEXT_ALIGN_CENTER,

```

(continues on next page)

(continued from previous page)

```

)
annot.set_border(width=0.3, dashes=[2])
annot.update(text_color=blue, fill_color=gold)
print_descr(annot)

r = annot.rect + displ
annot = page.add_text_annot(r.tl, t1)
print_descr(annot)

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

pos = annot.rect.bl # next insertion point
page.insert_text(pos, underline, morph=(pos, fitz.Matrix(-10)))
rl = page.search_for(underline, quads=True)
annot = page.add_underline_annot(rl[0])
print_descr(annot)

pos = annot.rect.bl
page.insert_text(pos, strikeout, morph=(pos, fitz.Matrix(-15)))
rl = page.search_for(strikeout, quads=True)
annot = page.add_strikeout_annot(rl[0])
print_descr(annot)

pos = annot.rect.bl
page.insert_text(pos, squiggled, morph=(pos, fitz.Matrix(-20)))
rl = page.search_for(squiggled, quads=True)
annot = page.add_squiggly_annot(rl[0])
print_descr(annot)

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

r += displ
annot = page.add_polygon_annot([r.bl, r.tr, r.br, r.tl]) # 'Polygon'
annot.set_border(width=0.3, dashes=[2])
annot.set_colors(stroke=blue, fill=gold)

```

(continues on next page)

(continued from previous page)

```

annot.set_line_ends(fitz.PDF_ANNOT_LE_DIAMOND, fitz.PDF_ANNOT_LE_CIRCLE)
annot.update()
print_descr(annot)

r += displ
annot = page.add_line_annot(r.tr, r.bl) # 'Line'
annot.set_border(width=0.3, dashes=[2])
annot.set_colors(stroke=blue, fill=gold)
annot.set_line_ends(fitz.PDF_ANNOT_LE_DIAMOND, fitz.PDF_ANNOT_LE_CIRCLE)
annot.update()
print_descr(annot)

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

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

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

r += displ
annot = page.add_stamp_annot(r, stamp=10) # 'Stamp'
annot.set_colors(stroke=green)
annot.update()
print_descr(annot)

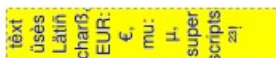
r += displ + (0, 0, 50, 10)
rc = page.insert_textbox(
    r,
    "This content will be removed upon applying the redaction.",
    color=blue,
    align=fitz.TEXT_ALIGN_CENTER,
)
annot = page.add_redact_annot(r)
print_descr(annot)

doc.save(__file__.replace(".py", "-%i.pdf" % page.rotation), deflate=True)

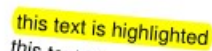
```

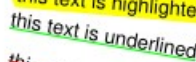
This script should lead to the following output:

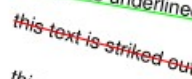
 Caret annotation

 FreeText annotation

 Text annotation

 Highlight annotation

 Underline annotation


 StrikeOut annotation

 Squiggly annotation

 PolyLine annotation

 Polygon annotation

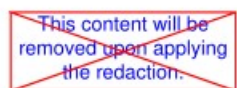
 Line annotation

 Square annotation

 Circle annotation

 FileAttachment annotation

 Stamp annotation

 Redact annotation

4.3.2 How to Use FreeText

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

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

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

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

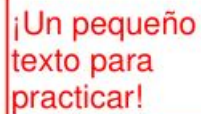
# 3 rectangles, same size, above each other
r1 = fitz.Rect(100,100,200,150)
r2 = r1 + (0,75,0,75)
r3 = r2 + (0,75,0,75)

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

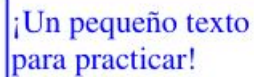
# add 3 annots, modify the last one somewhat
a1 = page.add_freetext_annot(r1, t, color=red)
a2 = page.add_freetext_annot(r2, t, fontname="Ti", color=blue)
a3 = page.add_freetext_annot(r3, t, fontname="Co", color=blue, rotate=90)
a3.set_border(width=0)
a3.update(fontsize=8, fill_color=gold)

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

The result looks like this:



¡Un pequeño
texto para
practicar!



¡Un pequeño texto
para practicar!



¡Un
pequeño
texto
para
practicar!

4.3.3 Using Buttons and JavaScript

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

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

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

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

Function 'this.getAnnots()' will return all annotations as an array. We loop
over this array to set the properties of the text through the 'richContents'
attribute.
There is no explicit property to set text to bold, but it is possible to set
fontWeight=800 (400 is the normal size) of richContents.
Other attributes, like color, italics, etc. can also be set via richContents.

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

Use PyMuPDF v1.16.12 to create a push button that executes a Javascript
containing the desired code. This is what this program does.
Then open the resulting file with Adobe reader (!).
After clicking on the button, all Freetext annotations will be bold, and the
file can be saved.
```

(continues on next page)

(continued from previous page)

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

Note / Caution:

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

"""

import sys

import fitz

this JavaScript will execute when the button is clicked:

jscript = """

var annt = this.getAnnots();

annt.forEach(function (item, index) {

 try {

 var span = item.richContents;

 span.forEach(function (it, dx) {

 it.fontWeight = 800;

 })

 item.richContents = span;

 } catch (err) {}

});

app.alert('Done');

"""

i_fn = sys.argv[1] *# input file name*

o_fn = "bold-" + i_fn *# output filename*

doc = fitz.open(i_fn) *# open input*

page = doc[0] *# get desired page*

-----

make a push button for invoking the JavaScript

-----

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

make it a 'PushButton'

widget.field_type = fitz.PDF_WIDGET_TYPE_BUTTON

widget.field_flags = fitz.PDF_BTN_FIELD_IS_PUSHBUTTON

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

widget.script = jscript *# fill in JavaScript source text*

widget.field_name = "Make bold" *# arbitrary name*

widget.field_value = "Off" *# arbitrary value*

widget.fill_color = (0, 0, 1) *# make button visible*

annot = page.add_widget(widget) *# add the widget to the page*

doc.save(o_fn) *# output the file*

4.3.4 How to Use Ink Annotations

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

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

```
import math
import fitz

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

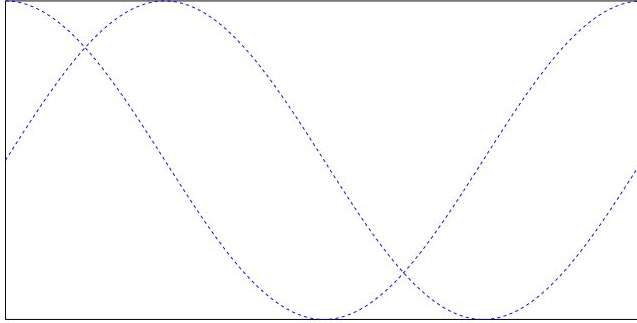
#-----
# create the document with one page
#-----
doc = fitz.open() # make new PDF
page = doc.new_page() # give it a page

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

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

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

This is the result:



4.4 Recipes: Drawing and Graphics

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

The syntax for such operations is defined in “A Operator Summary” on page 643 of the *Adobe PDF References*. Specifying these operators for a PDF page happens in its *contents* objects.

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

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

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

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

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

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

@author: Jorj
@license: GNU AFFERO GPL V3

Create a list of available symbols defined in shapes_and_symbols.py

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

"""

import fitz
import shapes_and_symbols as sas
```

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

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

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

```

This is the script's outcome:

- ▶ arrow (easy)
- ♦ caro (easy)
- ♣ clover (easy)
- ◆ diamond (easy)
- ⊘ do not enter (medium)
- ☹ frowney (medium)
- 👉 hand (complex)
- ♥ heart (easy)
- ✎ pencil (very complex)
- 😊 smiley (easy)

4.4.1 Extracting Drawings

- New in v1.18.0

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

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

The *path* dictionary has been designed such that it can easily be used by the [Shape](#) class and its methods. Here is an example for a page with one path, that draws a red-bordered yellow circle inside rectangle *Rect(100, 100, 200, 200)*:

```
>>> pprint(page.get_drawings())
[{'closePath': True,
 'color': [1.0, 0.0, 0.0],
 'dashes': '[] 0',
 'even_odd': False,
 'fill': [1.0, 1.0, 0.0],
 'items': [('c',
              Point(100.0, 150.0),
              Point(100.0, 177.614013671875),
              Point(122.38600158691406, 200.0),
              Point(150.0, 200.0)),
            ('c',
              Point(150.0, 200.0),
              Point(177.61399841308594, 200.0),
```

(continues on next page)

(continued from previous page)

```

        Point(200.0, 177.614013671875),
        Point(200.0, 150.0)),
        ('c',
        Point(200.0, 150.0),
        Point(200.0, 122.385986328125),
        Point(177.61399841308594, 100.0),
        Point(150.0, 100.0)),
        ('c',
        Point(150.0, 100.0),
        Point(122.38600158691406, 100.0),
        Point(100.0, 122.385986328125),
        Point(100.0, 150.0))),
'lineCap': (0, 0, 0),
'lineJoin': 0,
'opacity': 1.0,
'rect': Rect(100.0, 100.0, 200.0, 200.0),
'width': 1.0}]
>>>

```

Note: You need (at least) 4 Bézier curves (of 3rd order) to draw a circle with acceptable precision. See this [Wikipedia article](#) for some background.

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

```

import fitz
doc = fitz.open("some.file")
page = doc[0]
paths = page.get_drawings() # extract existing drawings
# this is a list of "paths", which can directly be drawn again using Shape
# -----
#
# define some output page with the same dimensions
outpdf = fitz.open()
outpage = outpdf.new_page(width=page.rect.width, height=page.rect.height)
shape = outpage.new_shape() # make a drawing canvas for the output page
# -----
# loop through the paths and draw them
# -----
for path in paths:
    # -----
    # draw each entry of the 'items' list
    # -----
    for item in path["items"]: # these are the draw commands
        if item[0] == "l": # line
            shape.draw_line(item[1], item[2])
        elif item[0] == "re": # rectangle
            shape.draw_rect(item[1])
        elif item[0] == "qu": # quad
            shape.draw_quad(item[1])
        elif item[0] == "c": # curve
            shape.draw_bezier(item[1], item[2], item[3], item[4])

```

(continues on next page)

(continued from previous page)

```

else:
    raise ValueError("unhandled drawing", item)
# -----
# all items are drawn, now apply the common properties
# to finish the path
# -----
shape.finish(
    fill=path["fill"], # fill color
    color=path["color"], # line color
    dashes=path["dashes"], # line dashing
    even_odd=path.get("even_odd", True), # control color of overlaps
    closePath=path["closePath"], # whether to connect last and first point
    lineJoin=path["lineJoin"], # how line joins should look like
    lineCap=max(path["lineCap"]), # how line ends should look like
    width=path["width"], # line width
    stroke_opacity=path.get("stroke_opacity", 1), # same value for both
    fill_opacity=path.get("fill_opacity", 1), # opacity parameters
)
# all paths processed - commit the shape to its page
shape.commit()
outpdf.save("drawings-page-0.pdf")

```

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

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

MuPDF v1.10 Changes and their implications for PyMuPDF

Pixmap

The alpha channel is now optional. Its presence is controlled by a new boolean parameter (called `alpha`). This has the following consequences:

- The size of one pixel can be two different values. For e.g. colorspace RGB, this size may be 3 (no alpha) or 4 bytes. The size of a pixmap is therefore determined not only by its colorspace, but also by its alpha value.
- Handling of pixmaps needs to take the alpha parameter into account. An decision has to be made during pixmap creation. Pixmaps coming from other sources or generated by some functions may or may not contain an alpha, which therefore needs to be checked.
- In general, the alpha channel should be avoided to benefit from significant memory savings.
- The `alpha` parameter disappeared: all methods saving pixmaps (`save_pixmap`) and friends) now always store the complete pixmap.
- Pixmaps created from PDF internal images may or may not contain an alpha - this solely depends on how the image was stored in the PDF.
- The `Pixmap` and `Image` classes have been extended with properties that help determine their characteristics - see the following list:
 - `Pixmap.alpha` - bool
 - `Pixmap.bytes` - integer containing the number of bytes of one line of the pixmap's effect
 - `Pixmap.height` - integer containing the number of bytes per pixel
 - `Pixmap.width` - integer containing the number of bytes used to determine the color (1 = devicecolor, 3 = devicecolor, 4 = devicecolor)
 - `Pixmap.colors` - string naming the colorspace, one of `devicecolor`, `devicecolor`, `devicecolor`
 - `Pixmap.colorspace` - `Colorspace` class
 - `Pixmap.colors` - `Colorspace` class

PyMuPDF Design Decision

Where PyMuPDF constructors require an alpha parameter, we assume `alpha = False` by default.

API Change: Display List

Constructor now requires the page's metabox.

API Change: Text Page

Constructor now requires the page's metabox.

API Change: Links

This contains significant changes:

- Link destinations objects are no longer maintained by MuPDF and information about destinations in general has been reduced. The only two variables containing such information now are `destinations` (a dict) and `links`, a string.
- The base class for PyMuPDF's `Link` (`Link`) has been deleted from MuPDF. In order to maintain backward compatibility, PyMuPDF provides an own `Link` class from available information as closely as possible.
- Document `links` now additionally also contains `links`, `links` and `links` properties.
- The following shows MuPDF's behavior concerning links and how we interpret this in PyMuPDF.

Internal	url	Links	Outlines
True	Starts with <code>destinations</code>	If <code>url</code> ends with a page number (format <code>destinations</code>), generate a <code>Link</code> , else as a <code>Link</code> . Strip off prefix and suffix in any case.	Behave as described in Links column. Page must be 0.

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

- Page definitions can be complex and include instructions for not showing / hiding certain areas to keep them

invisible. Things like this are ignored by `Page.get_drawings()` - it will always return all paths.

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

4.5 Recipes: Multiprocessing

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

No Python threading support.

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

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

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

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

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

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

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

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

    Notes:
        The PyMuPDF document cannot be part of the argument, because that
        cannot be pickled. So we are being passed in just its filename.
        This is no performance issue, because we are a separate process and
        need to open the document anyway.
```

(continues on next page)

(continued from previous page)

Any page-specific function can be processed here - rendering is just an example - text extraction might be another. The work must however be self-contained: no inter-process communication or synchronization is possible with this design. Care must also be taken with which parameters are contained in the argument, because it will be passed in via pickling by the Pool class. So any large objects will increase the overall duration.

Args:

vector: a list containing required parameters.

"""

recreate the arguments

idx = vector[0] # this is the segment number we have to process

cpu = vector[1] # number of CPUs

filename = vector[2] # document filename

mat = vector[3] # the matrix for rendering

doc = fitz.open(filename) # open the document

num_pages = doc.page_count # get number of pages

pages per segment: make sure that cpu * seg_size >= num_pages!

seg_size = int(num_pages / cpu + 1)

seg_from = idx * seg_size # our first page number

seg_to = min(seg_from + seg_size, num_pages) # last page number

for i in range(seg_from, seg_to): # work through our page segment

page = doc[i]

page.get_text("rawdict") # use any page-related type of work here, eg

pix = page.get_pixmap(alpha=False, matrix=mat)

store away the result somewhere ...

pix.save("p-%i.png" % i)

print("Processed page numbers %i through %i" % (seg_from, seg_to - 1))

if __name__ == "__main__":

t0 = mytime() # start a timer

filename = sys.argv[1]

mat = fitz.Matrix(0.2, 0.2) # the rendering matrix: scale down to 20%

cpu = cpu_count()

make vectors of arguments for the processes

vectors = [(i, cpu, filename, mat) for i in range(cpu)]

print("Starting %i processes for '%s'." % (cpu, filename))

pool = Pool() # make pool of 'cpu_count()' processes

pool.map(render_page, vectors, 1) # start processes passing each a vector

t1 = mytime() # stop the timer

print("Total time %g seconds" % round(t1 - t0, 2))

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

"""

(continues on next page)

(continued from previous page)

*Created on 2019-05-01**@author: yinkaisheng@live.com**@copyright: 2019 yinkaisheng@live.com**@license: GNU AFFERO GPL 3.0**Demonstrate the use of multiprocessing with PyMuPDF**-----
This example shows some more advanced use of multiprocessing.**The main process show a Qt GUI and establishes a 2-way communication with another process, which accesses a supported document.**"""*

```
import os
import sys
import time
import multiprocessing as mp
import queue
import fitz
```

*""" PyQt and PySide namespace unifier shim**<https://www.pythonguis.com/faq/pyqt6-vs-pyside6/>**simple "if 'PyQt6' in sys.modules:" test fails for me, so the more complex pkgutil use overkill for most people who might have one or the other, why both?**"""*

```
from pkgutil import iter_modules
```

```
def module_exists(module_name):
```

```
    return module_name in (name for loader, name, ispkg in iter_modules())
```

```
if module_exists("PyQt6"):
```

```
    # PyQt6
```

```
    from PyQt6 import QtGui, QtWidgets, QtCore
```

```
    from PyQt6.QtCore import pyqtSignal as Signal, pyqtSlot as Slot
```

```
    wrapper = "PyQt6"
```

```
elif module_exists("PySide6"):
```

```
    # PySide6
```

```
    from PySide6 import QtGui, QtWidgets, QtCore
```

```
    from PySide6.QtCore import Signal, Slot
```

```
    wrapper = "PySide6"
```

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

```
class DocForm(QtWidgets.QWidget):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.process = None
```

```
        self.queNum = mp.Queue()
```

```
        self.queDoc = mp.Queue()
```

(continues on next page)

(continued from previous page)

```

self.page_count = 0
self.curPageNum = 0
self.lastDir = ""
self.timerSend = QtCore.QTimer(self)
self.timerSend.timeout.connect(self.onTimerSendPageNum)
self.timerGet = QtCore.QTimer(self)
self.timerGet.timeout.connect(self.onTimerGetPage)
self.timerWaiting = QtCore.QTimer(self)
self.timerWaiting.timeout.connect(self.onTimerWaiting)
self.initUI()

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

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

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

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

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

    vbox.addLayout(hbox)

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

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

def openDoc(self):
    path, _ = QtWidgets.QFileDialog.getOpenFileName(
        self,
        "Open Document",
        self.lastDir,
        "All Supported Files (*.pdf;*.epub;*.xps;*.oxps;*.cbz;*.fb2);;PDF Files (*.

```

(continues on next page)

(continued from previous page)

```

→pdf));EPUB Files (*.epub));XPS Files (*.xps));OpenXPS Files (*.oxps));CBZ Files (*.
→cbz));FB2 Files (*.fb2)",
    #options=QtWidgets.QFileDialog.Options(),
)
if path:
    self.lastDir, self.file = os.path.split(path)
    if self.process:
        self.queueNum.put(-1) # use -1 to notify the process to exit
    self.timerSend.stop()
    self.curPageNum = 0
    self.page_count = 0
    self.process = mp.Process(
        target=openDocInProcess, args=(path, self.queueNum, self.queueDoc)
    )
    self.process.start()
    self.timerGet.start(40)
    self.label.setText("0/0")
    self.queueNum.put(0)
    self.startTime = time.perf_counter()
    self.timerWaiting.start(40)

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

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

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

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

```

(continues on next page)

(continued from previous page)

```

        pass

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

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

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

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

```

4.6 Recipes: General

4.6.1 How to Open with a Wrong File Extension

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

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

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

Note: MuPDF itself does not try to determine the file type from the file contents. **You** are responsible for supplying

the filetype info in some way – either implicitly via the file extension, or explicitly as shown. There are pure Python packages like `filetype` that help you doing this. Also consult the [Document](#) chapter for a full description.

If MuPDF encounters a file with an unknown / missing extension, it will try to open it as a PDF. So in these cases there is no need to for additional precautions. Similarly, for memory documents, you can just specify `doc=fitz.open(stream=mem_area)` to open it as a PDF document.

4.6.2 How to Embed or Attach Files

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

1. Attached Files: data are **attached to a page** by way of a *FileAttachment* annotation with this statement: `annot = page.add_file_annot(pos, ...)`, for details see [Page.add_file_annot\(\)](#). 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.embfile_add\(\)](#).

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

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

Also look at the sections above and at chapter Appendix 3.

4.6.3 How to Delete and Re-Arrange Pages

With PyMuPDF you have all options to copy, move, delete or re-arrange the pages of a PDF. Intuitive methods exist that allow you to do this on a page-by-page level, like the [Document.copy_page\(\)](#) 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.page_count`). The following lists represent the even and the odd page numbers, respectively:

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

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

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

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

The following example will reverse the order of all pages (**extremely fast**: sub-second time for the 756 pages of the *Adobe PDF References*):

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

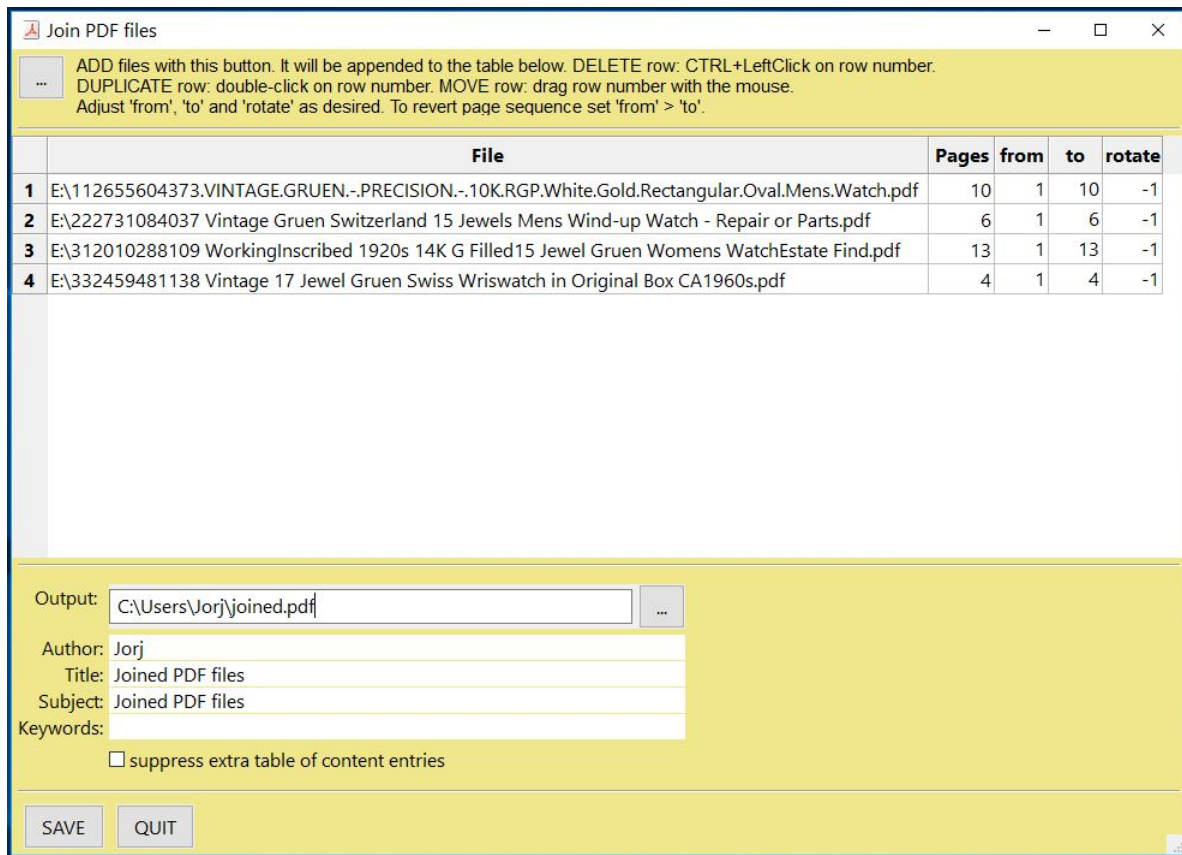
This snippet duplicates the PDF with itself so that it will contain the pages $0, 1, \dots, n, 0, 1, \dots, n$ (**extremely fast and without noticeably increasing the file size!**):

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

4.6.4 How to Join PDFs

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

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



4.6.5 How to Add Pages

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

`new_page`

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

```
>>> doc = fitz.open(...) # some new or existing PDF document
>>> page = doc.new_page(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.new_page()`. 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.paper_size("letter-l") # 'Letter' landscape
>>> page = doc.new_page(width = w, height = h)
```

The convenience function `paper_size()` knows over 40 industry standard paper formats to choose from. To see them, inspect dictionary `paperSizes`. Pass the desired dictionary key to `paper_size()` 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.new_page()
>>> doc.save("A4.pdf")
```

`insert_page`

`Document.insert_page()` 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.insert_page(to = -1, # default insertion point
                      text = None, # string or sequence of strings
                      fontsize = 11,
                      width = 595,
                      height = 842,
                      fontname = "Helvetica", # default font
                      fontfile = None, # any font file name
                      color = (0, 0, 0)) # text color (RGB)
```

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

4.6.6 How To Dynamically Clean Up Corrupt PDFs

This shows a potential use of PyMuPDF with another Python PDF library (the excellent pure Python package `pdfrw` 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

    # either we need a password or it is a problem-PDF
    # create a repaired / decompressed / decrypted version
    doc = fitz.open("pdf", ibuffer)
    if password is not None: # decrypt if password provided
        rc = doc.authenticate(password)
        if not rc > 0:
            raise ValueError("wrong password")
    c = doc.tobytes(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 for Windows only, but reported to also run under `Wine`) a similar result can be achieved, see [here](#). However, you must invoke it as a separate process via `subprocess.Popen`, using stdin and stdout as communication vehicles.

4.6.7 How to Split Single Pages

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

```

"""
Create a PDF copy with split-up pages (posterize)
-----
License: GNU AFFERO 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
"""

import fitz, sys
infile = sys.argv[1] # input file name
src = fitz.open(infile)
doc = fitz.open() # empty output PDF

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

    #-----
    # example: cut input page into 2 x 2 parts
    #-----
    r1 = r / 2 # 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.new_page(-1, # new output page with rx dimensions
                           width = rx.width,
```

(continues on next page)

(continued from previous page)

```

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

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

```

This shows what happens to an input page:



4.6.8 How to Combine Single Pages

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

```

'''
Copy an input PDF to output combining every 4 pages
-----
License: GNU AFFERO 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

```

(continues on next page)

(continued from previous page)

make the output page dimension dependent on input, or whatever.

Dependencies

PyMuPDF 1.12.1 or later

"""

```
import fitz, sys
infile = sys.argv[1]
src = fitz.open(infile)
doc = fitz.open() # empty output PDF

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

# define the 4 rectangles per page
r1 = r / 2 # top left rect
r2 = r1 + (r1.width, 0, r1.width, 0) # top right
r3 = r1 + (0, r1.height, 0, r1.height) # bottom left
r4 = fitz.Rect(r1.br, r.br) # bottom right

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

# now copy input pages to output
for spage in src:
    if spage.number % 4 == 0: # create new output page
        page = doc.new_page(-1,
                             width = width,
                             height = height)

        # insert input page into the correct rectangle
        page.show_pdf_page(r_tab[spage.number % 4], # select output rect
                           src, # input document
                           spage.number) # input page number

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

Example effect:



4.6.9 How to Convert Any Document to PDF

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

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

```

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

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

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

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

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

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

doc = fitz.open(fn)

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

toc= doc.get_toc() # table of contents of input
pdf.set_toc(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.get_pdf_now()
meta["creationDate"] = meta["modDate"]
pdf.set_metadata(meta)

```

(continues on next page)

(continued from previous page)

```

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

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

```

4.6.10 How to Deal with Messages Issued by MuPDF

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

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

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

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

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

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

```

>>> import fitz
>>> doc = fitz.open("damaged-file.pdf") # leads to a sys.stderr message:
mupdf: cannot find startxref
>>> print(fitz.TOOLS.mupdf_warnings()) # check if there is more info:
cannot find startxref
trying to repair broken xref
repairing PDF document
object missing 'endobj' token
>>> doc.can_save_incrementally() # this is to be expected:
False
>>> # the following indicates whether there are updates so far
>>> # this is the case because of the repair actions:

```

(continues on next page)

(continued from previous page)

```
>>> doc.is_dirty
True
>>> # the document has nevertheless been created:
>>> doc
fitz.Document('damaged-file.pdf')
>>> # we now know that any save must occur to a new file
```

Example output for an **unrecoverable error**:

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

4.6.11 How to Deal with PDF Encryption

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

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

Note: A PDF document may have two different passwords:

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

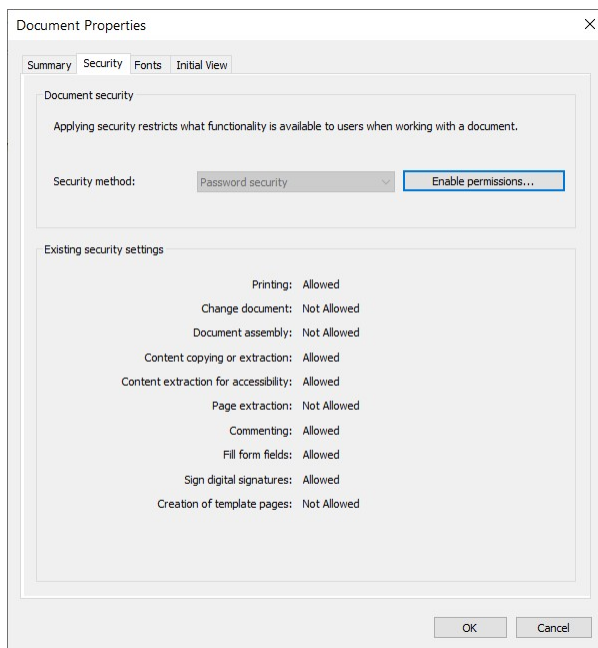
Method `Document.authenticate()` will automatically establish access rights according to the password used.

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

```
import fitz

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

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



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

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

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

4.7 Recipes: Common Issues and their Solutions

4.7.1 Changing Annotations: Unexpected Behaviour

4.7.1.1 Problem

There are two scenarios:

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

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

4.7.1.2 Cause

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

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

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

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

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

4.7.1.3 Solutions

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

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

4.7.2 Misplaced Item Insertions on PDF Pages

4.7.2.1 Problem

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

4.7.2.2 Cause

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

The visible image of a PDF page is controlled by commands coded in a special mini-language. For an overview of this language consult “Operator Summary” on pp. 643 of the [Adobe PDF References](#). These commands are stored in *contents* objects as strings (*bytes* in PyMuPDF).

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

So the PDF creator did this:

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

where they should have done this:

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

Note:

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

4.7.2.3 Solutions

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

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

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

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

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

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

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

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

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

4.7.3 Missing or Unreadable Extracted Text

Fairly often, text extraction does not work text as you would expect: text may be missing, or may not appear in the reading sequence visible on your screen, or contain garbled characters (like a ? or a "TOFU" symbol), etc. This can be caused by a number of different problems.

4.7.3.1 Problem: no text is extracted

Your PDF viewer does display text, but you cannot select it with your cursor, and text extraction delivers nothing.

4.7.3.2 Cause

1. You may be looking at an image embedded in the PDF page (e.g. a scanned PDF).
2. The PDF creator used no font, but **simulated** text by painting it, using little lines and curves. E.g. a capital “D” could be painted by a line “|” and a left-open semi-circle, an “o” by an ellipse, and so on.

4.7.3.3 Solution

Use an OCR software like [OCRmyPDF](#) to insert a hidden text layer underneath the visible page. The resulting PDF should behave as expected.

4.7.3.4 Problem: unreadable text

Text extraction does not deliver the text in readable order, duplicates some text, or is otherwise garbled.

4.7.3.5 Cause

1. The single characters are readable as such (no “<?” symbols), but the sequence in which the text is **coded in the file** deviates from the reading order. The motivation behind may be technical or protection of data against unwanted copies.
2. Many “<?” symbols occur, indicating MuPDF could not interpret these characters. The font may indeed be unsupported by MuPDF, or the PDF creator may have used a font that displays readable text, but on purpose obfuscates the originating corresponding unicode character.

4.7.3.6 Solution

1. Use layout preserving text extraction: `python -m fitz gettext file.pdf`.
2. If other text extraction tools also don’t work, then the only solution again is OCRing the page.

4.8 Recipes: 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.

It is a matter of documentation only in which chapter of the documentation you find what you are looking for. Everything is available and always via the same interface.

4.8.1 How to Iterate through the xref Table

A PDF's *xref* table is a list of all objects defined in the file. This table may easily contain many thousands of entries – the manual *Adobe PDF References* for example has 127,000 objects. Table entry “0” is reserved and must not be touched. The following script loops through the *xref* table and prints each object's definition:

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

This produces the following output:

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

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

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

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

A PDF object definition is an ordinary ASCII string.

4.8.2 How to Handle Object Streams

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

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

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

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

`Document.xref_stream()` automatically returns a stream decompressed as a bytes object – and `Document.update_stream()` automatically compresses it if beneficial.

4.8.3 How to Handle Page Contents

A PDF page can have zero or multiple `contents` objects. These are stream objects describing **what** appears **where** and **how** 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 643 of the *Adobe PDF References*.

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

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

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

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

For example, PyMuPDF adds new, small `contents` objects in methods `Page.insert_image()`, `Page.show_pdf_page()` and the `Shape` methods.

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

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

```
>>> # method 1: use the MuPDF clean function
>>> page.clean_contents() # cleans and combines multiple Contents
>>> xref = page.get_contents()[0] # only one /Contents now!
>>> cont = doc.xref_stream(xref)
>>> # this has also reformatted the PDF commands

>>> # method 2: extract concatenated contents
```

(continues on next page)

(continued from previous page)

```
>>> cont = page.read_contents()
>>> # the /Contents source itself is unmodified
```

The clean function `Page.clean_contents()` does a lot more than just glueing `contents` objects: it also corrects and optimizes the PDF operator syntax of the page and removes any inconsistencies with the page's object definition.

4.8.4 How to Access the PDF Catalog

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

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

Note: Indentation, line breaks and comments are inserted here for clarification purposes only and will not normally appear. For more information on the PDF catalog see section 7.7.2 on page 71 of the *Adobe PDF References*.

4.8.5 How to Access the PDF File Trailer

The trailer of a PDF file is a *dictionary* located towards the end of the file. It contains special objects, and pointers to important other information. See *Adobe PDF References* p. 42. Here is an overview:

Key	Type	Value
Size	int	Number of entries in the cross-reference table + 1.
Prev	int	Offset to previous <i>xref</i> section (indicates incremental updates).
Root	dictionary	(indirect) Pointer to the catalog. See previous section.
Encrypt	dictionary	Pointer to encryption object (encrypted files only).
Info	dictionary	(indirect) Pointer to information (metadata).
ID	array	File identifier consisting of two byte strings.
XRefStm	int	Offset of a cross-reference stream. See <i>Adobe PDF References</i> p. 49.

Access this information via PyMuPDF with `Document.pdf_trailer()` or, equivalently, via `Document.xref_object()` using -1 instead of a valid *xref* number.

```
>>> import fitz
>>> doc=fitz.open("PyMuPDF.pdf")
>>> print(doc.xref_object(-1)) # or: print(doc.pdf_trailer())
<<
/Type /XRef
/Index [ 0 8263 ]
/Size 8263
/W [ 1 3 1 ]
/Root 8260 0 R
/Info 8261 0 R
/ID [ <4339B9CEE46C2CD28A79EBDDD67CC9B3> <4339B9CEE46C2CD28A79EBDDD67CC9B3> ]
/Length 19883
/Filter /FlateDecode
>>
>>>
```

4.8.6 How to Access XML Metadata

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

PyMuPDF has no way to **interpret or change** this information directly, because it contains no XML features. XML metadata is however stored as a *stream* object, so it can be read, modified with appropriate software and written back.

```
>>> xmlmetadata = doc.get_xml_metadata()
>>> print(xmlmetadata)
<?xpacket begin="\uffeff" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="3.1-702">
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
...
omitted data
...
<?xpacket end="w"?>
```

Using some XML package, the XML data can be interpreted and / or modified and then stored back. The following also works, if the PDF previously had no XML metadata:

```
>>> # write back modified XML metadata:
>>> doc.set_xml_metadata(xmlmetadata)
>>>
>>> # XML metadata can be deleted like this:
>>> doc.del_xml_metadata()
```


4.8.7 How to Extend PDF Metadata

Attribute `Document.metadata` is designed so it works for all supported document types in the same way: it is a Python dictionary with a **fixed set of key-value pairs**. Correspondingly, `Document.set_metadata()` only accepts standard keys.

However, PDFs may contain items not accessible like this. Also, there may be reasons to store additional information, like copyrights. Here is a way to handle **arbitrary metadata items** by using PyMuPDF low-level functions.

As an example, look at this standard metadata output of some PDF:

```
# -----
# standard metadata
# -----
pprint(doc.metadata)
{'author': 'PRINCE',
 'creationDate': "D:2010102417034406'-30'",
 'creator': 'PrimoPDF http://www.primopdf.com/',
 'encryption': None,
 'format': 'PDF 1.4',
 'keywords': '',
 'modDate': "D:20200725062431-04'00'",
 'producer': 'macOS Version 10.15.6 (Build 19G71a) Quartz PDFContext, '
             'AppendMode 1.1',
 'subject': '',
 'title': 'Full page fax print',
 'trapped': ''}
```

Use the following code to see **all items** stored in the metadata object:

```
# -----
# metadata including private items
# -----
metadata = {} # make my own metadata dict
what, value = doc.xref_get_key(-1, "Info") # /Info key in the trailer
if what != "xref":
    pass # PDF has no metadata
else:
    xref = int(value.replace("0 R", "")) # extract the metadata xref
    for key in doc.xref_get_keys(xref):
        metadata[key] = doc.xref_get_key(xref, key)[1]
pprint(metadata)
{'Author': 'PRINCE',
 'CreationDate': "D:2010102417034406'-30'",
 'Creator': 'PrimoPDF http://www.primopdf.com/',
 'ModDate': "D:20200725062431-04'00'",
 'PXCViewerInfo': 'PDF-XChange Viewer;2.5.312.1;Feb 9 '
                  '2015;12:00:06;D:20200725062431-04'00'',
 'Producer': 'macOS Version 10.15.6 (Build 19G71a) Quartz PDFContext, '
             'AppendMode 1.1',
 'Title': 'Full page fax print'}
# -----
# note the additional 'PXCViewerInfo' key - ignored in standard!
# -----
```

Vice versa, you can also **store private metadata items** in a PDF. It is your responsibility to make sure that these items conform to PDF specifications - especially they must be (unicode) strings. Consult section 14.3 (p. 548) of the *Adobe PDF References* for details and caveats:

```
what, value = doc.xref_get_key(-1, "Info") # /Info key in the trailer
if what != "xref":
    raise ValueError("PDF has no metadata")
xref = int(value.replace("0 R", "")) # extract the metadata xref
# add some private information
doc.xref_set_key(xref, "mykey", fitz.get_pdf_str(" is Beijing"))
#
# after executing the previous code snippet, we will see this:
pprint(metadata)
{'Author': 'PRINCE',
 'CreationDate': "D:2010102417034406'-30'",
 'Creator': 'PrimoPDF http://www.primopdf.com/',
 'ModDate': "D:20200725062431-04'00'",
 'PXCViewerInfo': 'PDF-XChange Viewer;2.5.312.1;Feb 9 '
                  '2015;12:00:06;D:20200725062431-04'00'',
 'Producer': 'macOS Version 10.15.6 (Build 19G71a) Quartz PDFContext, '
              'AppendMode 1.1',
 'Title': 'Full page fax print',
 'mykey': ' is Beijing'}
```

To delete selected keys, use `doc.xref_set_key(xref, "mykey", "null")`. As explained in the next section, string “null” is the PDF equivalent to Python’s `None`. A key with that value will be treated as not being specified – and physically removed in garbage collections.

4.8.8 How to Read and Update PDF Objects

There also exist granular, elegant ways to access and manipulate selected PDF *dictionary* keys.

- `Document.xref_get_keys()` returns the PDF keys of the object at `xref`:

```
In [1]: import fitz
In [2]: doc = fitz.open("pymupdf.pdf")
In [3]: page = doc[0]
In [4]: from pprint import pprint
In [5]: pprint(doc.xref_get_keys(page.xref))
('Type', 'Contents', 'Resources', 'MediaBox', 'Parent')
```

- Compare with the full object definition:

```
In [6]: print(doc.xref_object(page.xref))
<<
  /Type /Page
  /Contents 1297 0 R
  /Resources 1296 0 R
  /MediaBox [ 0 0 612 792 ]
  /Parent 1301 0 R
>>
```

- Single keys can also be accessed directly via `Document.xref_get_key()`. The value **always is a string** together with type information, that helps with interpreting it:

```
In [7]: doc.xref_get_key(page.xref, "MediaBox")
Out[7]: ('array', '[0 0 612 792]')
```

- Here is a full listing of the above page keys:

```
In [9]: for key in doc.xref_get_keys(page.xref):
...:     print("%s = %s" % (key, doc.xref_get_key(page.xref, key)))
...:
Type = ('name', '/Page')
Contents = ('xref', '1297 0 R')
Resources = ('xref', '1296 0 R')
MediaBox = ('array', '[0 0 612 792]')
Parent = ('xref', '1301 0 R')
```

- An undefined key inquiry returns `('null', 'null')` – PDF object type `null` corresponds to `None` in Python. Similar for the booleans `true` and `false`.
- Let us add a new key to the page definition that sets its rotation to 90 degrees (you are aware that there actually exists `Page.set_rotation()` for this?):

```
In [11]: doc.xref_get_key(page.xref, "Rotate") # no rotation set:
Out[11]: ('null', 'null')
In [12]: doc.xref_set_key(page.xref, "Rotate", "90") # insert a new key
In [13]: print(doc.xref_object(page.xref)) # confirm success
<<
  /Type /Page
  /Contents 1297 0 R
  /Resources 1296 0 R
  /MediaBox [ 0 0 612 792 ]
  /Parent 1301 0 R
  /Rotate 90
>>
```

- This method can also be used to remove a key from the `xref` dictionary by setting its value to `null`: The following will remove the rotation specification from the page: `doc.xref_set_key(page.xref, "Rotate", "null")`. Similarly, to remove all links, annotations and fields from a page, use `doc.xref_set_key(page.xref, "Annots", "null")`. Because `Annots` by definition is an array, setting an empty array with the statement `doc.xref_set_key(page.xref, "Annots", "[]")` would do the same job in this case.
- PDF dictionaries can be hierarchically nested. In the following page object definition both, `Font` and `XObject` are subdictionaries of `Resources`:

```
In [15]: print(doc.xref_object(page.xref))
<<
  /Type /Page
  /Contents 1297 0 R
  /Resources <<
    /XObject <<
      /Im1 1291 0 R
    >>
    /Font <<
      /F39 1299 0 R
```

(continues on next page)

(continued from previous page)

```

    /F40 1300 0 R
  >>
>>
/MediaBox [ 0 0 612 792 ]
/Parent 1301 0 R
/Rotate 90
>>

```

- The above situation **is supported** by methods `Document.xref_set_key()` and `Document.xref_get_key()`: use a path-like notation to point at the required key. For example, to retrieve the value of key `Im1` above, specify the complete chain of dictionaries “above” it in the key argument: `"Resources/XObject/Im1"`:

```

In [16]: doc.xref_get_key(page.xref, "Resources/XObject/Im1")
Out[16]: ('xref', '1291 0 R')

```

- The path notation can also be used to **directly set a value**: use the following to let `Im1` point to a different object:

```

In [17]: doc.xref_set_key(page.xref, "Resources/XObject/Im1", "9999 0 R")
In [18]: print(doc.xref_object(page.xref)) # confirm success:
<<
  /Type /Page
  /Contents 1297 0 R
  /Resources <<
    /XObject <<
      /Im1 9999 0 R
    >>
    /Font <<
      /F39 1299 0 R
      /F40 1300 0 R
    >>
  >>
  /MediaBox [ 0 0 612 792 ]
  /Parent 1301 0 R
  /Rotate 90
>>

```

Be aware, that **no semantic checks** whatsoever will take place here: if the PDF has no xref 9999, it won't be detected at this point.

- If a key does not exist, it will be created by setting its value. Moreover, if any intermediate keys do not exist either, they will also be created as necessary. The following creates an array `D` several levels below the existing dictionary `A`. Intermediate dictionaries `B` and `C` are automatically created:

```

In [5]: print(doc.xref_object(xref)) # some existing PDF object:
<<
  /A <<
  >>
>>
In [6]: # the following will create 'B', 'C' and 'D'
In [7]: doc.xref_set_key(xref, "A/B/C/D", "[1 2 3 4]")
In [8]: print(doc.xref_object(xref)) # check out what happened:
<<
  /A <<

```

(continues on next page)

(continued from previous page)

```

/B <<
  /C <<
    /D [ 1 2 3 4 ]
  >>
>>
>>
>>

```

- When setting key values, basic **PDF syntax checking** will be done by MuPDF. For example, new keys can only be created **below a dictionary**. The following tries to create some new string item E below the previously created array D:

```

In [9]: # 'D' is an array, no dictionary!
In [10]: doc.xref_set_key(xref, "A/B/C/D/E", "(hello)")
mupdf: not a dict (array)
--- ... ---
RuntimeError: not a dict (array)

```

- It is also **not possible**, to create a key if some higher level key is an “**indirect**” object, i.e. an xref. In other words, xrefs can only be modified directly and not implicitly via other objects referencing them:

```

In [13]: # the following object points to an xref
In [14]: print(doc.xref_object(4))
<<
  /E 3 0 R
>>
In [15]: # 'E' is an indirect object and cannot be modified here!
In [16]: doc.xref_set_key(4, "E/F", "90")
mupdf: path to 'F' has indirects
--- ... ---
RuntimeError: path to 'F' has indirects

```

Caution: These are expert functions! There are no validations as to whether valid PDF objects, xrefs, etc. are specified. As with other low-level methods there is the risk to render the PDF, or parts of it unusable.

4.9 Recipes: Journalling

Starting with version 1.19.0, journalling is possible when updating PDF documents.

Journalling is a logging mechanism which permits either **reverting** or **re-applying** changes to a PDF. Similar to LUWs “Logical Units of Work” in modern database systems, one can group a set of updates into an “operation”. In MuPDF journalling, an operation plays the role of a LUW.

Note: In contrast to LUW implementations found in database systems, MuPDF journalling happens on a **per document level**. There is no support for simultaneous updates across multiple PDFs: one would have to establish one’s own logic here.

- Journalling must be *enabled* via a document method. Journalling is possible for existing or new documents. Journalling **can be disabled only** by closing the file.
- Once enabled, every change must happen inside an *operation* – otherwise an exception is raised. An operation is started and stopped via document methods. Updates happening between these two calls form an LUW and can thus collectively be rolled back or re-applied, or, in MuPDF terminology “undo” resp. “redo”.
- At any point, the journalling status can be queried: whether journalling is active, how many operations have been recorded, whether “undo” or “redo” is possible, the current position inside the journal, etc.
- The journal can be **saved to** or **loaded from** a file. These are document methods.
- When loading a journal file, compatibility with the document is checked and journalling is automatically enabled upon success.
- For an **existing** PDF being journalled, a special new save method is available: `Document.save_snapshot()`. This performs a special incremental save that includes all journalled updates so far. If its journal is saved at the same time (immediately after the document snapshot), then document and journal are in sync and can later on be used together to undo or redo operations or to continue journalled updates – just as if there had been no interruption.
- The snapshot PDF is a valid PDF in every aspect and fully usable. If the document is however changed in any way without using its journal file, then a desynchronization will take place and the journal is rendered unusable.
- Snapshot files are structured like incremental updates. Nevertheless, the internal journalling logic requires, that saving **must happen to a new file**. So the user should develop a file naming convention to support recognizable relationships between an original PDF, like `original.pdf` and its snapshot sets, like `original-snap1.pdf` / `original-snap1.log`, `original-snap2.pdf` / `original-snap2.log`, etc.

4.9.1 Example Session 1

Description:

- Make a new PDF and enable journalling. Then add a page and some text lines – each as a separate operation.
- Navigate within the journal, undoing and redoing these updates and displaying status and file results:

```
>>> import fitz
>>> doc=fitz.open()
>>> doc.journal_enable()

>>> # try update without an operation:
>>> page = doc.new_page()
mupdf: No journalling operation started
... omitted lines
RuntimeError: No journalling operation started

>>> doc.journal_start_op("op1")
>>> page = doc.new_page()
>>> doc.journal_stop_op()

>>> doc.journal_start_op("op2")
>>> page.insert_text((100,100), "Line 1")
>>> doc.journal_stop_op()

>>> doc.journal_start_op("op3")
>>> page.insert_text((100,120), "Line 2")
```

(continues on next page)

(continued from previous page)

```

>>> doc.journal_stop_op()

>>> doc.journal_start_op("op4")
>>> page.insert_text((100,140), "Line 3")
>>> doc.journal_stop_op()

>>> # show position in journal
>>> doc.journal_position()
(4, 4)
>>> # 4 operations recorded - positioned at bottom
>>> # what can we do?
>>> doc.journal_can_do()
{'undo': True, 'redo': False}
>>> # currently only undos are possible. Print page content:
>>> print(page.get_text())
Line 1
Line 2
Line 3

>>> # undo last insert:
>>> doc.journal_undo()
>>> # show combined status again:
>>> doc.journal_position();doc.journal_can_do()
(3, 4)
{'undo': True, 'redo': True}
>>> print(page.get_text())
Line 1
Line 2

>>> # our position is now second to last
>>> # last text insertion was reverted
>>> # but we can redo / move forward as well:
>>> doc.journal_redo()
>>> # our combined status:
>>> doc.journal_position();doc.journal_can_do()
(4, 4)
{'undo': True, 'redo': False}
>>> print(page.get_text())
Line 1
Line 2
Line 3
>>> # line 3 has appeared again!

```

4.9.2 Example Session 2

Description:

- Similar to previous, but after undoing some operations, we now add a different update. This will cause:
 - permanent removal of the undone journal entries
 - the new update operation will become the new last entry.

```
>>> doc=fitz.open()
>>> doc.journal_enable()
>>> doc.journal_start_op("Page insert")
>>> page=doc.new_page()
>>> doc.journal_stop_op()
>>> for i in range(5):
>>>     doc.journal_start_op("insert-%i" % i)
>>>     page.insert_text((100, 100 + 20*i), "text line %i" %i)
>>>     doc.journal_stop_op()
```

```
>>> # combined status info:
>>> doc.journal_position();doc.journal_can_do()
(6, 6)
{'undo': True, 'redo': False}
```

```
>>> for i in range(3): # revert last three operations
>>>     doc.journal_undo()
>>> doc.journal_position();doc.journal_can_do()
(3, 6)
{'undo': True, 'redo': True}
```

```
>>> # now do a different update:
>>> doc.journal_start_op("Draw some line")
>>> page.draw_line((100,150), (300,150))
Point(300.0, 150.0)
>>> doc.journal_stop_op()
>>> doc.journal_position();doc.journal_can_do()
(4, 4)
{'undo': True, 'redo': False}
```

```
>>> # this has changed the journal:
>>> # previous last 3 text line operations were removed, and
>>> # we have only 4 operations: drawing the line is the new last one
```


4.10 Recipes: Stories

This document showcases some typical use cases for *Stories*.

As mentioned in the *tutorial*, stories may be created using up to three input sources: HTML, CSS and Archives – all of which are optional and which, respectively, can be provided programmatically.

The following examples will showcase combinations for using these inputs.

Note: Many of these recipe’s source code are included as examples in the docs folder.

4.10.1 How to add a line of text with some formatting

Here is the inevitable “Hello World” example. We will show two variants:

1. Create using existing HTML source¹, that may come from anywhere.
 2. Create using the Python API.
-

Variant using an existing HTML source¹ – which in this case is defined as a constant in the script:

```
import fitz

HTML = """
<p style="font-family: sans-serif;color: blue">Hello World!</p>
"""

MEDIABOX = fitz.paper_rect("letter") # output page format: Letter
WHERE = MEDIABOX + (36, 36, -36, -36) # leave borders of 0.5 inches

story = fitz.Story(html=HTML) # create story from HTML
writer = fitz.DocumentWriter("output.pdf") # create the writer

more = 1 # will indicate end of input once it is set to 0

while more: # loop outputting the story
    device = writer.begin_page(MEDIABOX) # make new page
    more, _ = story.place(WHERE) # layout into allowed rectangle
    story.draw(device) # write on page
    writer.end_page() # finish page

writer.close() # close output file
```

¹ HTML & CSS support

Note: At the time of writing the HTML engine for Stories is fairly basic and supports a subset of CSS2 attributes.

Some important CSS support to consider:

- The only available layout is relative layout.
- background is unavailable, use background-color instead.
- float is unavailable.

Note: The above effect (sans-serif and blue text) could have been achieved by using a separate CSS source like so:

```
import fitz

CSS = """
body {
    font-family: sans-serif;
    color: blue;
}
"""

HTML = """
<p>Hello World!</p>
"""

# the story would then be created like this:
story = fitz.Story(html=HTML, user_css=CSS)
```

The Python API variant – everything is created programmatically:

```
import fitz

MEDIABOX = fitz.paper_rect("letter")
WHERE = MEDIABOX + (36, 36, -36, -36)

story = fitz.Story() # create an empty story
body = story.body # access the body of its DOM
with body.add_paragraph() as para: # store desired content
    para.set_font("sans-serif").set_color("blue").add_text("Hello World!")

writer = fitz.DocumentWriter("output.pdf")

more = 1

while more:
    device = writer.begin_page(MEDIABOX)
    more, _ = story.place(WHERE)
    story.draw(device)
    writer.end_page()

writer.close()
```

Both variants will produce the same output PDF.

4.10.2 How to use images

Images can be referenced in the provided HTML source, or the reference to a desired image can also be stored via the Python API. In any case, this requires using an *Archive*, which refers to the place where the image can be found.

Note: Images with the binary content embedded in the HTML source are **not supported** by stories.

We extend our “Hello World” example from above and display an image of our planet right after the text. Assuming the image has the name “world.jpg” and is present in the script’s folder, then this is the modified version of the above Python API variant:

```
import fitz

MEDIABOX = fitz.paper_rect("letter")
WHERE = MEDIABOX + (36, 36, -36, -36)

# create story, let it look at script folder for resources
story = fitz.Story(archive=".")
body = story.body # access the body of its DOM

with body.add_paragraph() as para:
    # store desired content
    para.set_font("sans-serif").set_color("blue").add_text("Hello World!")

# another paragraph for our image:
with body.add_paragraph() as para:
    # store image in another paragraph
    para.add_image("world.jpg")

writer = fitz.DocumentWriter("output.pdf")

more = 1

while more:
    device = writer.begin_page(MEDIABOX)
    more, _ = story.place(WHERE)
    story.draw(device)
    writer.end_page()

writer.close()
```

4.10.3 Reading external HTML and CSS for a Story

These cases are fairly straightforward.

As a general recommendation, HTML and CSS sources should be **read as binary files** and decoded before using them in a story. The Python `pathlib.Path` provides convenient ways to do this:

```
import pathlib
import fitz

htmlpath = pathlib.Path("myhtml.html")
csspath = pathlib.Path("mycss.css")

HTML = htmlpath.read_bytes().decode()
CSS = csspath.read_bytes().decode()

story = fitz.Story(html=HTML, user_css=CSS)
```

4.10.4 How to output database content with Story templates

This script demonstrates how to report SQL database content using an **HTML template**.

The example SQL database contains two tables:

1. Table “films” contains one row per film with the fields “**title**”, “**director**” and (release) “**year**”.
2. Table “actors” contains one row per actor and film title (fields (actor) “**name**” and (film) “**title**”).

The story DOM consists of a template for one film, which reports film data together with a list of casted actors.

Files:

- docs/samples/filmfestival-sql.py
- docs/samples/filmfestival-sql.db

```
"""
This is a demo script for using PyMuPDF with its "Story" feature.

The following aspects are being covered here:

* The script produces a report of films that are stored in an SQL database
* The report format is provided as a HTML template

The SQL database contains two tables:
1. Table "films" which has the columns "title" (film title, str), "director"
   (str) and "year" (year of release, int).
2. Table "actors" which has the columns "name" (actor name, str) and "title"
   (the film title where the actor had been casted, str).

The script reads all content of the "films" table. For each film title it
reads all rows from table "actors" which took part in that film.
```

(continues on next page)

(continued from previous page)

Comment 1

To keep things easy and free from pesky technical detail, the relevant file names inherit the name of this script:

- the database's filename is the script name with ".py" extension replaced by ".db".
- the output PDF similarly has script file name with extension ".pdf".

Comment 2

The SQLITE database has been created using <https://sqlitebrowser.org/>, a free multi-platform tool to maintain or manipulate SQLITE databases.

"""

import os**import sqlite3****import fitz**

HTML template for the film report

There are four placeholders coded as "id" attributes.

One "id" allows locating the template part itself, the other three

indicate where database text should be inserted.

festival_template = (

"<html><head><title>Just some arbitrary text</title></head>"

'<body><h1 style="text-align:center">Hook Norton Film Festival</h1>'

""

'<li id="filmtemplate">'

'<b id="filmtitle">'

"<dl>"

'<dt>Director<dd id="director">'

'<dt>Release Year<dd id="filmyear">'

'<dt>Cast<dd id="cast">'

"</dl>"

""

""

"</body></html>"

)

define database access

dbfilename = __file__.replace(".py", ".db") # the SQLITE database file name

assert os.path.isfile(dbfilename), f'{dbfilename}'

database = sqlite3.connect(dbfilename) # open database

cursor_films = database.cursor() # cursor for selecting the films

cursor_casts = database.cursor() # cursor for selecting actors per film

select statement for the films - let SQL also sort it for us

select_films = """SELECT title, director, year FROM films ORDER BY title"""

(continues on next page)

(continued from previous page)

```

# select stament for actors, a skeleton: sub-select by film title
select_casts = """SELECT name FROM actors WHERE film = "%s" ORDER BY name"""

# -----
# define the HTML Story and fill it with database data
# -----
story = fitz.Story(festival_template)
body = story.body # access the HTML body detail
template = body.find(None, "id", "filmtemplate") # find the template part

# read the films from the database and put them all in one Python list
# NOTE: instead we might fetch rows one by one (advisable for large volumes)
cursor_films.execute(select_films) # execute cursor, and ...
films = cursor_films.fetchall() # read out what was found

for title, director, year in films: # iterate through the films
    film = template.clone() # clone template to report each film
    film.find(None, "id", "filmtitle").add_text(title) # put title in templ
    film.find(None, "id", "director").add_text(director) # put director
    film.find(None, "id", "filmyear").add_text(str(year)) # put year

    # the actors reside in their own table - find the ones for this film title
    cursor_casts.execute(select_casts % title) # execute cursor
    casts = cursor_casts.fetchall() # read actors for the film
    # each actor name appears in its own tuple, so extract it from there
    film.find(None, "id", "cast").add_text("\n".join([c[0] for c in casts]))
    body.append_child(film)

template.remove() # remove the template

# -----
# generate the PDF
# -----
writer = fitz.DocumentWriter(__file__.replace(".py", ".pdf"), "compress")
mediabox = fitz.paper_rect("a4") # use pages in ISO-A4 format
where = mediabox + (72, 36, -36, -72) # leave page borders

more = 1 # end of output indicator

while more:
    dev = writer.begin_page(mediabox) # make a new page
    more, filled = story.place(where) # arrange content for this page
    story.draw(dev, None) # write content to page
    writer.end_page() # finish the page

writer.close() # close the PDF

```

4.10.5 How to integrate with existing PDFs

Because a *DocumentWriter* can only write to a new file, stories cannot be placed on existing pages. This script demonstrates a circumvention of this restriction.

The basic idea is letting *DocumentWriter* output to a PDF in memory. Once the story has finished, we re-open this memory PDF and put its pages to desired locations on **existing** pages via method *Page.show_pdf_page()*.

Files:

- docs/samples/showpdf-page.py

```

"""
Demo of Story class in PyMuPDF
-----

This script demonstrates how to the results of a fitz.Story output can be
placed in a rectangle of an existing (!) PDF page.

"""
import io
import os

import fitz

def make_pdf(fileptr, text, rect, font="sans-serif", archive=None):
    """Make a memory DocumentWriter from HTML text and a rect.

    Args:
        fileptr: a Python file object. For example an io.BytesIO().
        text: the text to output (HTML format)
        rect: the target rectangle. Will use its width / height as mediabox
        font: (str) font family name, default sans-serif
        archive: fitz.Archive parameter. To be used if e.g. images or special
            fonts should be used.

    Returns:
        The matrix to convert page rectangles of the created PDF back
        to rectangle coordinates in the parameter "rect".
        Normal use will expect to fit all the text in the given rect.
        However, if an overflow occurs, this function will output multiple
        pages, and the caller may decide to either accept or retry with
        changed parameters.
    """
    # use input rectangle as the page dimension
    mediabox = fitz.Rect(0, 0, rect.width, rect.height)
    # this matrix converts mediabox back to input rect
    matrix = mediabox.torect(rect)

    story = fitz.Story(text, archive=archive)
    body = story.body
    body.set_properties(font=font)
    writer = fitz.DocumentWriter(fileptr)

```

(continues on next page)

(continued from previous page)

```

while True:
    device = writer.begin_page(mediabox)
    more, _ = story.place(mediabox)
    story.draw(device)
    writer.end_page()
    if not more:
        break
writer.close()
return matrix

# -----
# We want to put this in a given rectangle of an existing page
# -----
HTML = """
<p>PyMuPDF is a great package! And it still improves significantly from one version to
↳ the next one!</p>
<p>It is a Python binding for <b>MuPDF</b>, a lightweight PDF, XPS, and E-book viewer,
↳ renderer, and toolkit.<br> Both are maintained and developed by Artifex Software, Inc.
↳</p>
<p>Via MuPDF it can access files in PDF, XPS, OpenXPS, CBZ, EPUB, MOBI and FB2 (e-books)
↳ formats,<br> and it is known for its top
<b><i>performance</i></b> and <b><i>rendering quality.</i></b></p>"""

# Make a PDF page for demo purposes
root = os.path.abspath( f"{__file__}/.." )
doc = fitz.open(f"{root}/mupdf-title.pdf")
page = doc[0]

WHERE = fitz.Rect(50, 100, 250, 500) # target rectangle on existing page

fileptr = io.BytesIO() # let DocumentWriter use this as its file

# -----
# call DocumentWriter and Story to fill our rectangle
matrix = make_pdf(fileptr, HTML, WHERE)
# -----
src = fitz.open("pdf", fileptr) # open DocumentWriter output PDF
if src.page_count > 1: # target rect was too small
    raise ValueError("target WHERE too small")

# its page 0 contains our result
page.show_pdf_page(WHERE, src, 0)

doc.ez_save(f"{root}/mupdf-title-after.pdf")

```


4.10.6 How to make multi-columnned layouts and access fonts from package `pymupdf-fonts`

This script outputs an article (taken from Wikipedia) that contains text and multiple images and uses a 2-column page layout.

In addition, two “Ubuntu” font families from package `pymupdf-fonts` are used instead of defaulting to Base-14 fonts.

Yet another feature used here is that all data – the images and the article HTML – are jointly stored in a ZIP file.

Files:

- `docs/samples/quickfox.py`
- `docs/samples/quickfox.zip`

```
"""
This is a demo script using PyMuPDF's Story class to output text as a PDF with
a two-column page layout.

The script demonstrates the following features:
* How to fill columns or table cells of complex page layouts
* How to embed images
* How to modify existing, given HTML sources for output (text indent, font size)
* How to use fonts defined in package "pymupdf-fonts"
* How to use ZIP files as Archive

-----
The example is taken from the somewhat modified Wikipedia article
https://en.wikipedia.org/wiki/The_quick_brown_fox_jumps_over_the_lazy_dog.
-----
"""

import io
import os
import zipfile
import fitz

thisdir = os.path.dirname(os.path.abspath(__file__))
myzip = zipfile.ZipFile(os.path.join(thisdir, "quickfox.zip"))
arch = fitz.Archive(myzip)

if fitz.fitz_fontdescriptors:
    # we want to use the Ubuntu fonts for sans-serif and for monospace
    CSS = fitz.css_for_pymupdf_font("ubuntu", archive=arch, name="sans-serif")
    CSS = fitz.css_for_pymupdf_font("ubuntm", CSS=CSS, archive=arch, name="monospace")
else:
    # No pymupdf-fonts available.
    CSS=""

docname = __file__.replace(".py", ".pdf") # output PDF file name

HTML = myzip.read("quickfox.html").decode()
```

(continues on next page)

(continued from previous page)

```

# make the Story object
story = fitz.Story(HTML, user_css=CSS, archive=arch)

# -----
# modify the DOM somewhat
# -----
body = story.body # access HTML body
body.set_properties(font="sans-serif") # and give it our font globally

# modify certain nodes
para = body.find("p", None, None) # find relevant nodes (here: paragraphs)
while para != None:
    para.set_properties( # method MUST be used for existing nodes
        indent=15,
        fontsize=13,
    )
    para = para.find_next("p", None, None)

# choose PDF page size
MEDIABOX = fitz.paper_rect("letter")
# text appears only within this subrectangle
WHERE = MEDIABOX + (36, 36, -36, -36)

# -----
# define page layout within the WHERE rectangle
# -----
COLS = 2 # layout: 2 cols 1 row
ROWS = 1
TABLE = fitz.make_table(WHERE, cols=COLS, rows=ROWS)
# fill the cells of each page in this sequence:
CELLS = [TABLE[i][j] for i in range(ROWS) for j in range(COLS)]

fileobject = io.BytesIO() # let DocumentWriter write to memory
writer = fitz.DocumentWriter(fileobject) # define the writer

more = 1
while more: # loop until all input text has been written out
    dev = writer.begin_page(MEDIABOX) # prepare a new output page
    for cell in CELLS:
        # content may be complete after any cell, ...
        if more: # so check this status first
            more, _ = story.place(cell)
            story.draw(dev)
    writer.end_page() # finish the PDF page

writer.close() # close DocumentWriter output

# for housekeeping work re-open from memory
doc = fitz.open("pdf", fileobject)
doc.ez_save(docname)

```

4.10.7 How make a layout which wraps around a predefined “no go area” layout

This is a demo script using PyMuPDF’s Story class to output text as a PDF with a two-column page layout.

The script demonstrates the following features:

- Layout text around images of an existing (“target”) PDF.
- Based on a few global parameters, areas on each page are identified, that can be used to receive text layouted by a Story.
- These global parameters are not stored anywhere in the target PDF and must therefore be provided in some way:
 - The width of the border(s) on each page.
 - The fontsize to use for text. This value determines whether the provided text will fit in the empty spaces of the (fixed) pages of target PDF. It cannot be predicted in any way. The script ends with an exception if target PDF has not enough pages, and prints a warning message if not all pages receive at least some text. In both cases, the FONTSIZE value can be changed (a float value).
 - Use of a 2-column page layout for the text.
- The layout creates a temporary (memory) PDF. Its produced page content (the text) is used to overlay the corresponding target page. If text requires more pages than are available in target PDF, an exception is raised. If not all target pages receive at least some text, a warning is printed.
- The script reads “image-no-go.pdf” in its own folder. This is the “target” PDF. It contains 2 pages with each 2 images (from the original article), which are positioned at places that create a broad overall test coverage. Otherwise the pages are empty.
- The script produces “quickfox-image-no-go.pdf” which contains the original pages and image positions, but with the original article text laid out around them.

Files:

- docs/samples/quickfox-image-no-go.py
- docs/samples/quickfox-image-no-go.pdf
- docs/samples/quickfox.zip

```

"""
This is a demo script using PyMuPDF's Story class to output text as a PDF with
a two-column page layout.

The script demonstrates the following features:
* Layout text around images of an existing ("target") PDF.
* Based on a few global parameters, areas on each page are identified, that
  can be used to receive text layouted by a Story.
* These global parameters are not stored anywhere in the target PDF and
  must therefore be provided in some way.
  - The width of the border(s) on each page.
  - The fontsize to use for text. This value determines whether the provided
    text will fit in the empty spaces of the (fixed) pages of target PDF. It
    cannot be predicted in any way. The script ends with an exception if

```

(continues on next page)

(continued from previous page)

```

    target PDF has not enough pages, and prints a warning message if not all
    pages receive at least some text. In both cases, the FONTSIZE value
    can be changed (a float value).
- Use of a 2-column page layout for the text.
* The layout creates a temporary (memory) PDF. Its produced page content
  (the text) is used to overlay the corresponding target page. If text
  requires more pages than are available in target PDF, an exception is raised.
  If not all target pages receive at least some text, a warning is printed.
* The script reads "image-no-go.pdf" in its own folder. This is the "target" PDF.
  It contains 2 pages with each 2 images (from the original article), which are
  positioned at places that create a broad overall test coverage. Otherwise the
  pages are empty.
* The script produces "quickfox-image-no-go.pdf" which contains the original pages
  and image positions, but with the original article text laid out around them.

```

Note:

```

-----
This script version uses just image positions to derive "No-Go areas" for
layouting the text. Other PDF objects types are detectable by PyMuPDF and may
be taken instead or in addition, without influencing the layouting.
The following are candidates for other such "No-Go areas". Each can be detected
and located by PyMuPDF:
* Annotations
* Drawings
* Existing text

```

```

-----
The text and images are taken from the somewhat modified Wikipedia article
https://en.wikipedia.org/wiki/The\_quick\_brown\_fox\_jumps\_over\_the\_lazy\_dog.
-----

```

```

"""

```

```

import io
import os
import zipfile
import fitz

```

```

thisdir = os.path.dirname(os.path.abspath(__file__))
myzip = zipfile.ZipFile(os.path.join(thisdir, "quickfox.zip"))

```

```

docname = os.path.join(thisdir, "image-no-go.pdf") # "no go" input PDF file name
outname = os.path.join(thisdir, "quickfox-image-no-go.pdf") # output PDF file name
BORDER = 36 # global parameter
FONTSIZE = 12.5 # global parameter
COLS = 2 # number of text columns, global parameter

```

```

def analyze_page(page):
    """Compute MediaBox and rectangles on page that are free to receive text.

```

Notes:

(continues on next page)

(continued from previous page)

Assume a *BORDER* around the page, make 2 columns of the resulting sub-rectangle and extract the rectangles of all images on page. For demo purposes, the image rectangles are taken as "NO-GO areas" on the page when writing text with the Story. The function returns free areas for each of the columns.

Returns:

""" (page.number, mediabox, CELLS), where CELLS is a list of free cells. """

```

prect = page.rect # page rectangle - will be our MEDIABOX later
where = prect + (BORDER, BORDER, -BORDER, -BORDER)
TABLE = fitz.make_table(where, rows=1, cols=COLS)

# extract rectangles covered by images on this page
IMG_RECTS = sorted( # image rects on page (sort top-left to bottom-right)
    [fitz.Rect(item["bbox"]) for item in page.get_image_info()],
    key=lambda b: (b.y1, b.x0),
)

def free_cells(column):
    """Return free areas in this column."""
    free_stripes = [] # y-value pairs wrapping a free area stripe
    # intersecting images: block complete intersecting column stripe
    col_imgs = [(b.y0, b.y1) for b in IMG_RECTS if abs(b & column) > 0]
    s_y0 = column.y0 # top y-value of column
    for y0, y1 in col_imgs: # an image stripe
        if y0 > s_y0 + FONTSIZE: # image starts below last free btm value
            free_stripes.append((s_y0, y0)) # store as free stripe
            s_y0 = y1 # start of next free stripe

    if s_y0 + FONTSIZE < column.y1: # enough room to column bottom
        free_stripes.append((s_y0, column.y1))

    if free_stripes == []: # covers "no image in this column"
        free_stripes.append((column.y0, column.y1))

    # make available cells of this column
    CELLS = [fitz.Rect(column.x0, y0, column.x1, y1) for (y0, y1) in free_stripes]
    return CELLS

# collection of available Story rectangles on page
CELLS = []
for i in range(COLS):
    CELLS.extend(free_cells(TABLE[0][i]))

return page.number, prect, CELLS

HTML = myzip.read("quickfox.html").decode()

# -----
# Make the Story object

```

(continues on next page)

(continued from previous page)

```

# -----
story = fitz.Story(HTML)

# modify the DOM somewhat
body = story.body # access HTML body
body.set_properties(font="sans-serif") # and give it our font globally

# modify certain nodes
para = body.find("p", None, None) # find relevant nodes (here: paragraphs)
while para != None:
    para.set_properties( # method MUST be used for existing nodes
        indent=15,
        fontsize=FONTSIZE,
    )
    para = para.find_next("p", None, None)

# we remove all image references, because the target PDF already has them
img = body.find("img", None, None)
while img != None:
    next_img = img.find_next("img", None, None)
    img.remove()
    img = next_img

page_info = {} # contains MEDIABOX and free CELLS per page
doc = fitz.open(docname)
for page in doc:
    pno, mediabox, cells = analyze_page(page)
    page_info[pno] = (mediabox, cells)
doc.close() # close target PDF for now - re-open later

fileobject = io.BytesIO() # let DocumentWriter write to memory
writer = fitz.DocumentWriter(fileobject) # define output writer

more = 1 # stop if this ever becomes zero
pno = 0 # count output pages
while more: # loop until all HTML text has been written
    try:
        MEDIABOX, CELLS = page_info[pno]
    except KeyError: # too much text space required: reduce fontsize?
        raise ValueError("text does not fit on target PDF")
    dev = writer.begin_page(MEDIABOX) # prepare a new output page
    for cell in CELLS: # iterate over free cells on this page
        if not more: # need to check this for every cell
            continue
        more, _ = story.place(cell)
        story.draw(dev)
    writer.end_page() # finish the PDF page
    pno += 1

writer.close() # close DocumentWriter output

# Re-open writer output, read its pages and overlay target pages with them.

```

(continues on next page)

(continued from previous page)

```

# The generated pages have same dimension as their targets.
src = fitz.open("pdf", fileobject)
doc = fitz.open(doc.name)
for page in doc: # overlay every target page with the prepared text
    if page.number >= src.page_count:
        print(f"Text only uses {src.page_count} target pages!")
        continue # story did not need all target pages?

    # overlay target page
    page.show_pdf_page(page.rect, src, page.number)

    # DEBUG start --- draw the text rectangles
    # mb, cells = page_info[page.number]
    # for cell in cells:
    #     page.draw_rect(cell, color=(1, 0, 0))
    # DEBUG stop ---

doc.ez_save(outname)

```

4.10.8 How to output a table

Support for HTML tables is yet not complete in MuPDF. It is however possible to output tables with equal column widths that do not cross page boundaries.

This script reflects existing features.

Files:

- docs/samples/table01.py

```

import fitz

table_text = (
    (
        "Length",
        "integer",
        """(Required) The number of bytes from the beginning of the line following the
→keyword stream to the last byte just before the keyword endstream. (There may be an
→additional EOL marker, preceding endstream, that is not included in the count and is
→not logically part of the stream data.) See "Stream Extent," above, for further
→discussion."""),
    ),
    (
        "Filter",
        "name or array",
        """(Optional) The name of a filter to be applied in processing the stream data
→found between the keywords stream and endstream, or an array of such names. Multiple
→filters should be specified in the order in which they are to be applied."""),
    )

```

(continues on next page)

(continued from previous page)

```

    ),
    (
        "FFilter",
        "name or array",
        """(Optional; PDF 1.2) The name of a filter to be applied in processing the data.
        ↳ found in the stream's external file, or an array of such names. The same rules apply as
        ↳ for Filter."""),
    ),
)

HTML = """
<html>
<body><h2>TABLE 3.4 Entries common to all stream dictionaries</h2>
<table style="width: 100%">
    <tr>
        <th class="w25">KEY
        <th class="w25">TYPE
        <th class="w50">VALUE
    </tr>
    <tr id="rowtemplate">
        <td id="col0" class="w25"></td>
        <td id="col1" class="w25"></td>
        <td id="col2" class="w50"></td>
    </tr>
"""
CSS = """
body {font-family: sans-serif;}
th {text-align: left;}
td {font-size: 8px;}
.w25 {width: 50px;}
.w50 {width: 300px;}
"""

story = fitz.Story(HTML, user_css=CSS)
body = story.body
template = body.find(None, "id", "rowtemplate")
parent = template.parent

for col0, col1, col2 in table_text:
    row = template.clone()
    row.find(None, "id", "col0").add_text("\n" + col0)
    row.find(None, "id", "col1").add_text("\n" + col1)
    row.find(None, "id", "col2").add_text("\n" + col2)
    parent.append_child(row)
template.remove()

writer = fitz.DocumentWriter(__file__.replace(".py", ".pdf"), "compress")
mediabox = fitz.paper_rect("letter")
where = mediabox + (36, 36, -36, -36)

more = 1
while more:

```

(continues on next page)

(continued from previous page)

```

dev = writer.begin_page(mediabox)
more, filled = story.place(where)
story.draw(dev, None)
writer.end_page()
writer.close()

```

4.10.9 How to create a simple grid layout

By creating a sequence of *Story* objects within a grid created via the *make_table* function a developer can create grid layouts as required.

Files:

- docs/samples/simple-grid.py

```

import fitz

MEDIABOX = fitz.paper_rect("letter") # output page format: Letter
GRIDSPACE = fitz.Rect(100, 100, 400, 400)
GRID = fitz.make_table(GRIDSPACE, rows=2, cols=2)
CELLS = [GRID[i][j] for i in range(2) for j in range(2)]
text_table = ("A", "B", "C", "D")
writer = fitz.DocumentWriter(__file__.replace(".py", ".pdf")) # create the writer

device = writer.begin_page(MEDIABOX) # make new page
for i, text in enumerate(text_table):
    story = fitz.Story(em=1)
    body = story.body
    with body.add_paragraph() as para:
        para.set_bgcolor("#ecc")
        para.set_pagebreak_after() # fills whole cell with bgcolor
        para.set_align("center")
        para.set_fontsize(16)
        para.add_text(f"\n\n\n{text}")
    story.place(CELLS[i])
    story.draw(device)
    del story

writer.end_page() # finish page

writer.close() # close output file

```

4.10.10 How to generate a Table of Contents

This script lists the source code of all Python scripts that live in the script's directory.

Files:

- docs/samples/code-printer.py

```

"""
Demo script PyMuPDF Story class
-----

Read the Python sources in the script directory and create a PDF of all their
source codes.

The following features are included as a specialty:
1. HTML source for fitz.Story created via Python API exclusively
2. Separate Story objects for page headers and footers
3. Use of HTML "id" elements for identifying source start pages
4. Generate a Table of Contents pointing to source file starts. This
   - uses the new Story callback feature
   - uses Story also for making the TOC page(s)

"""

import io
import os
import time

import fitz

THISDIR = os.path.dirname(os.path.abspath(__file__))
TOC = [] # this will contain the TOC list items
CURRENT_ID = "" # currently processed filename - stored by recorder func
MEDIABOX = fitz.paper_rect("a4-l") # chosen page size
WHERE = MEDIABOX + (36, 50, -36, -36) # sub rectangle for source content
# location of the header rectangle
HDR_WHERE = (36, 5, MEDIABOX.width - 36, 40)
# location of the footer rectangle
FTR_WHERE = (36, MEDIABOX.height - 36, MEDIABOX.width - 36, MEDIABOX.height)

def recorder(elpos):
    """Callback function invoked during story.place().
    This function generates / collects all TOC items and updates the value of
    CURRENT_ID - which is used to update the footer line of each page.
    """
    global TOC, CURRENT_ID
    if not elpos.open_close & 1: # only consider "open" items
        return
    level = elpos.heading
    y0 = elpos.rect[1] # top of written rectangle (use for TOC)
    if level > 0: # this is a header (h1 - h6)
        pno = elpos.page + 1 # the page number

```

(continues on next page)

(continued from previous page)

```

        TOC.append(
            (
                level,
                elpos.text,
                elpos.page + 1,
                y0,
            )
        )
    )
    return

CURRENT_ID = elpos.id if elpos.id else "" # update for footer line
return

def header_story(text):
    """Make the page header"""
    header = fitz.Story()
    hdr_body = header.body
    hdr_body.add_paragraph().set_properties(
        align=fitz.fitz.TEXT_ALIGN_CENTER,
        bgcolor="#eee",
        font="sans-serif",
        bold=True,
        fontsize=12,
        color="green",
    ).add_text(text)
    return header

def footer_story(text):
    """Make the page footer"""
    footer = fitz.Story()
    ftr_body = footer.body
    ftr_body.add_paragraph().set_properties(
        bgcolor="#eee",
        align=fitz.TEXT_ALIGN_CENTER,
        color="blue",
        fontsize=10,
        font="sans-serif",
    ).add_text(text)
    return footer

def code_printer(outfile):
    """Output the generated PDF to outfile."""
    global MAX_TITLE_LEN
    where = +WHERE
    writer = fitz.DocumentWriter(outfile, "")
    print_time = time.strftime("%Y-%m-%d %H:%M:%S (%z)")
    thispath = os.path.abspath(os.curdir)
    basename = os.path.basename(thispath)

```

(continues on next page)

(continued from previous page)

```

story = fitz.Story()
body = story.body
body.set_properties(font="sans-serif")

text = f"Python sources in folder '{THISDIR}'"

body.add_header(1).add_text(text)  # the only h1 item in the story

files = os.listdir(THISDIR)  # list / select Python files in our directory
i = 1
for code_file in files:
    if not code_file.endswith(".py"):
        continue

    # read Python file source
    fileinput = open(os.path.join(THISDIR, code_file), "rb")
    text = fileinput.read().decode()
    fileinput.close()

    # make level 2 header
    hdr = body.add_header(2)
    if i > 1:
        hdr.set_pagebreak_before()
    hdr.add_text(f"{i}. Listing of file '{code_file}'")

    # Write the file code
    body.add_codeblock().set_bgcolor((240, 255, 210)).set_color("blue").set_id(
        code_file
    ).set_fontsize(10).add_text(text)

    # Indicate end of a source file
    body.add_paragraph().set_align(fitz.TEXT_ALIGN_CENTER).add_text(
        f"----- End of File '{code_file}' -----"
    )
    i += 1  # update file counter

i = 0
while True:
    i += 1
    device = writer.begin_page(MEDIABOX)
    # create Story objects for header, footer and the rest.
    header = header_story(f"Python Files in '{THISDIR}'")
    hdr_ok, _ = header.place(HDR_WHERE)
    if hdr_ok != 0:
        raise ValueError("header does not fit")
    header.draw(device, None)

    # -----
    # Write the file content.
    # -----
    more, filled = story.place(where)
    # Inform the callback function

```

(continues on next page)

(continued from previous page)

```

# Args:
# recorder: the Python function to call
# {}: dictionary containing anything - we pass the page number
story.element_positions(recorder, {"page": i - 1})
story.draw(device, None)

# -----
# Make / write page footer.
# We MUST have a paragraph b/o background color / alignment
# -----
if CURRENT_ID:
    text = f"File '{CURRENT_ID}' printed at {print_time}{chr(160)*5}{'- '*10}"
    ↪{chr(160)*5}Page {i}"
else:
    text = f"Printed at {print_time}{chr(160)*5}{'- '*10}{chr(160)*5}Page {i}"
    footer = footer_story(text)
    # write the page footer
    ftr_ok, _ = footer.place(FTR_WHERE)
    if ftr_ok != 0:
        raise ValueError("footer does not fit")
    footer.draw(device, None)

writer.end_page()
if more == 0:
    break
writer.close()

if __name__ == "__main__" or os.environ.get('PYTEST_CURRENT_TEST'):
    fileptr1 = io.BytesIO()
    t0 = time.perf_counter()
    code_printer(fileptr1) # make the PDF
    t1 = time.perf_counter()
    doc = fitz.open("pdf", fileptr1)
    old_count = doc.page_count
    # -----
    # Post-processing step to make / insert the toc
    # This also works using fitz.Story:
    # - make a new PDF in memory which contains pages with the TOC text
    # - add these TOC pages to the end of the original file
    # - search item text on the inserted pages and cover each with a PDF link
    # - move the TOC pages to the front of the document
    # -----
    story = fitz.Story()
    body = story.body
    body.add_header(1).set_font("sans-serif").add_text("Table of Contents")
    # prefix TOC with an entry pointing to this page
    TOC.insert(0, [1, "Table of Contents", old_count + 1, 36])

    for item in TOC[1:]: # write the file name headers as TOC lines
        body.add_paragraph().set_font("sans-serif").add_text(
            item[1] + f" - ({item[2]})"

```

(continues on next page)

(continued from previous page)

```

    )
    fileptr2 = io.BytesIO() # put TOC pages to a separate PDF initially
    writer = fitz.DocumentWriter(fileptr2)
    i = 1
    more = 1
    while more:
        device = writer.begin_page(MEDIABOX)
        header = header_story(f"Python Files in '{THISDIR}'")
        # write the page header
        hdr_ok, _ = header.place(HDR_WHERE)
        header.draw(device, None)

        more, filled = story.place(WHERE)
        story.draw(device, None)

        footer = footer_story(f"TOC-{i}") # separate page numbering scheme
        # write the page footer
        ftr_ok, _ = footer.place(FTR_WHERE)
        footer.draw(device, None)
        writer.end_page()
        i += 1

    writer.close()
    doc2 = fitz.open("pdf", fileptr2) # open TOC pages as another PDF
    doc.insert_pdf(doc2) # and append to the main PDF
    new_range = range(old_count, doc.page_count) # the TOC page numbers
    pages = [doc[i] for i in new_range] # these are the TOC pages within main PDF
    for item in TOC: # search for TOC item text to get its rectangle
        for page in pages:
            rl = page.search_for(item[1], flags=~fitz.TEXT_PRESERVE_LIGATURES)
            if rl != []: # this text must be on next page
                break
        rect = rl[0] # rectangle of TOC item text
        link = { # make a link from it
            "kind": fitz.LINK_GOTO,
            "from": rect,
            "to": fitz.Point(0, item[3]),
            "page": item[2] - 1,
        }
        page.insert_link(link)

    # insert the TOC in the main PDF
    doc.set_toc(TOC)
    # move all the TOC pages to the desired place (1st page here)
    for i in new_range:
        doc.move_page(doc.page_count - 1, 0)
    doc.ez_save(__file__.replace(".py", ".pdf"))

```

It features the following capabilities:

- Automatic generation of a Table of Contents (TOC) on separately numbered pages at the start of the document - using a specialized *Story*.

- Use of 3 separate *Story* objects per page: header story, footer story and the story for printing the Python sources.
 - The page **footer is automatically changed** to show the name of the current Python file.
- Use of *Story.element_positions()* to collect the data for the TOC and for the dynamic adjustment of page footers. This is an example of a **bidirectional communication** between the story output process and the script.
- The main PDF with the Python sources is being written to memory by its *DocumentWriter*. Another *Story* / *DocumentWriter* pair is then used to create a (memory) PDF for the TOC pages. Finally, both these PDFs are joined and the result stored to disk.

4.10.11 How to display a list from JSON data

This example takes some JSON data input which it uses to populate a *Story*. It also contains some visual text formatting and shows how to add links.

Files:

- docs/samples/json-example.py

```
import fitz
import json

my_json = """
[
    {
        "name" :          "Five-storied Pagoda",
        "temple" :         "Rurikō-ji",
        "founded" :        "middle Muromachi period, 1442",
        "region" :         "Yamaguchi, Yamaguchi",
        "position" :        "34.190181,131.472917"
    },
    {
        "name" :          "Founder's Hall",
        "temple" :         "Eihō-ji",
        "founded" :        "early Muromachi period",
        "region" :         "Tajimi, Gifu",
        "position" :        "35.346144,137.129189"
    },
    {
        "name" :          "Fudōdō",
        "temple" :         "Kongōbu-ji",
        "founded" :        "early Kamakura period",
        "region" :         "Kōya, Wakayama",
        "position" :        "34.213103,135.580397"
    },
    {
        "name" :          "Goeidō",
        "temple" :         "Nishi Honganji",
        "founded" :        "Edo period, 1636",
        "region" :         "Kyoto",
        "position" :        "34.991394,135.751689"
    }
]
```

(continues on next page)

(continued from previous page)

```

},
{
    "name" :          "Golden Hall",
    "temple" :         "Murō-ji",
    "founded" :        "early Heian period",
    "region" :         "Uda, Nara",
    "position" :       "34.536586819357986,136.0395548452301"
},
{
    "name" :          "Golden Hall",
    "temple" :         "Fudō-in",
    "founded" :        "late Muromachi period, 1540",
    "region" :         "Hiroshima",
    "position" :       "34.427014,132.471117"
},
{
    "name" :          "Golden Hall",
    "temple" :         "Ninna-ji",
    "founded" :        "Momoyama period, 1613",
    "region" :         "Kyoto",
    "position" :       "35.031078,135.713811"
},
{
    "name" :          "Golden Hall",
    "temple" :         "Mii-dera",
    "founded" :        "Momoyama period, 1599",
    "region" :         "Ōtsu, Shiga",
    "position" :       "35.013403,135.852861"
},
{
    "name" :          "Golden Hall",
    "temple" :         "Tōshōdai-ji",
    "founded" :        "Nara period, 8th century",
    "region" :         "Nara, Nara",
    "position" :       "34.675619,135.784842"
},
{
    "name" :          "Golden Hall",
    "temple" :         "Tō-ji",
    "founded" :        "Momoyama period, 1603",
    "region" :         "Kyoto",
    "position" :       "34.980367,135.747686"
},
{
    "name" :          "Golden Hall",
    "temple" :         "Tōdai-ji",
    "founded" :        "middle Edo period, 1705",
    "region" :         "Nara, Nara",
    "position" :       "34.688992,135.839822"
},
{
    "name" :          "Golden Hall",

```

(continues on next page)

(continued from previous page)

```

    "temple" :      "Hōryū-ji",
    "founded" :     "Asuka period, by 693",
    "region" :      "Ikaruga, Nara",
    "position" :    "34.614317,135.734458"
  },
  {
    "name" :        "Golden Hall",
    "temple" :      "Daigo-ji",
    "founded" :     "late Heian period",
    "region" :      "Kyoto",
    "position" :    "34.951481,135.821747"
  },
  {
    "name" :        "Keigū-in Main Hall",
    "temple" :      "Kōryū-ji",
    "founded" :     "early Kamakura period, before 1251",
    "region" :      "Kyoto",
    "position" :    "35.015028,135.705425"
  },
  {
    "name" :        "Konpon-chūdō",
    "temple" :      "Enryaku-ji",
    "founded" :     "early Edo period, 1640",
    "region" :      "Ōtsu, Shiga",
    "position" :    "35.070456,135.840942"
  },
  {
    "name" :        "Korō",
    "temple" :      "Tōshōdai-ji",
    "founded" :     "early Kamakura period, 1240",
    "region" :      "Nara, Nara",
    "position" :    "34.675847,135.785069"
  },
  {
    "name" :        "Kōfūzō",
    "temple" :      "Hōryū-ji",
    "founded" :     "early Heian period",
    "region" :      "Ikaruga, Nara",
    "position" :    "34.614439,135.735428"
  },
  {
    "name" :        "Large Lecture Hall",
    "temple" :      "Hōryū-ji",
    "founded" :     "middle Heian period, 990",
    "region" :      "Ikaruga, Nara",
    "position" :    "34.614783,135.734175"
  },
  {
    "name" :        "Lecture Hall",
    "temple" :      "Zuiryū-ji",
    "founded" :     "early Edo period, 1655",
    "region" :      "Takaoka, Toyama",

```

(continues on next page)

(continued from previous page)

```

        "position" :      "36.735689,137.010019"
    },
    {
        "name" :          "Lecture Hall",
        "temple" :         "Tōshōdai-ji",
        "founded" :        "Nara period, 763",
        "region" :          "Nara, Nara",
        "position" :        "34.675933,135.784842"
    },
    {
        "name" :          "Lotus Flower Gate",
        "temple" :         "Tō-ji",
        "founded" :        "early Kamakura period",
        "region" :          "Kyoto",
        "position" :        "34.980678,135.746314"
    },
    {
        "name" :          "Main Hall",
        "temple" :         "Akishinodera",
        "founded" :        "early Kamakura period",
        "region" :          "Nara, Nara",
        "position" :        "34.703769,135.776189"
    }
]

"""

# the result is a Python dictionary:
my_dict = json.loads(my_json)

MEDIABOX = fitz.paper_rect("letter") # output page format: Letter
WHERE = MEDIABOX + (36, 36, -36, -36)
writer = fitz.DocumentWriter("json-example.pdf") # create the writer

story = fitz.Story()
body = story.body

for i, entry in enumerate(my_dict):

    for attribute, value in entry.items():
        para = body.add_paragraph()

        if attribute == "position":
            para.set_fontsize(10)
            para.add_link(f"www.google.com/maps/@{value},14z")
        else:
            para.add_span()
            para.set_color("#990000")
            para.set_fontsize(14)
            para.set_bold()
            para.add_text(f"{attribute} ")
            para.add_span()

```

(continues on next page)

(continued from previous page)

```

        para.set_fontsize(18)
        para.add_text(f"{value}")

    body.add_horizontal_line()

# This while condition will check a value from the Story `place` method
# for whether all content for the story has been written (0), otherwise
# more content is waiting to be written (1)
more = 1
while more:
    device = writer.begin_page(MEDIABOX) # make new page
    more, _ = story.place(WHERE)
    story.draw(device)
    writer.end_page() # finish page

writer.close() # close output file

del story

```

4.10.12 Using the alternative Story.write*() functions

The Story.write*() functions provide a different way to use the *Story* functionality, removing the need for calling code to implement a loop that calls *Story.place()* and *Story.draw()* etc, at the expense of having to provide at least a rectfn() callback.

4.10.12.1 How to do basic layout with Story.write()

This script lays out multiple copies of its own source code, into four rectangles per page.

Files:

- docs/samples/story-write.py

```

"""
Demo script for PyMuPDF's `Story.write()` method.

This is a way of laying out a story into a PDF document, that avoids the need
to write a loop that calls `story.place()` and `story.draw()`.

Instead just a single function call is required, albeit with a `rectfn()`
callback that returns the rectangles into which the story is placed.
"""

import html

import fitz

```

(continues on next page)

(continued from previous page)

```

# Create html containing multiple copies of our own source code.
#
with open(__file__) as f:
    text = f.read()
text = html.escape(text)
html = f'''
<!DOCTYPE html>
<body>

<h1>Contents of {__file__}</h1>

<h2>Normal</h2>
<pre>
{text}
</pre>

<h2>Strong</h2>
<strong>
<pre>
{text}
</pre>
</strong>

<h2>Em</h2>
<em>
<pre>
{text}
</pre>
</em>

</body>
'''

def rectfn(rect_num, filled):
    """
    We return four rectangles per page in this order:

        1 3
        2 4
    """
    page_w = 800
    page_h = 600
    margin = 50
    rect_w = (page_w - 3*margin) / 2
    rect_h = (page_h - 3*margin) / 2

    if rect_num % 4 == 0:
        # New page.
        mediabox = fitz.Rect(0, 0, page_w, page_h)

```

(continues on next page)

(continued from previous page)

```

else:
    mediabox = None
    # Return one of four rects in turn.
    rect_x = margin + (rect_w+margin) * ((rect_num // 2) % 2)
    rect_y = margin + (rect_h+margin) * (rect_num % 2)
    rect = fitz.Rect(rect_x, rect_y, rect_x + rect_w, rect_y + rect_h)
    #print(f'rectfn(): rect_num={rect_num} filled={filled}. Returning: rect={rect}')
    return mediabox, rect, None

story = fitz.Story(html, em=8)

out_path = __file__.replace('.py', '.pdf')
writer = fitz.DocumentWriter(out_path)

story.write(writer, rectfn)
writer.close()

```

4.10.12.2 How to do iterative layout for a table of contents with `Story.write_stabilized()`

This script creates html content dynamically, adding a contents section based on `ElementPosition` items that have non-zero `.heading` values.

The contents section is at the start of the document, so modifications to the contents can change page numbers in the rest of the document, which in turn can cause page numbers in the contents section to be incorrect.

So the script uses `Story.write_stabilized()` to repeatedly lay things out until things are stable.

Files:

- docs/samples/story-write-stabilized.py

```

"""
Demo script for PyMuPDF's `fitz.Story.write_stabilized()`.

`fitz.Story.write_stabilized()` is similar to `fitz.Story.write()`,
except instead of taking a fixed html document, it does iterative layout
of dynamically-generated html content (provided by a callback) to a
`fitz.DocumentWriter`.

For example this allows one to add a dynamically-generated table of contents
section while ensuring that page numbers are patched up until stable.
"""

import textwrap

import fitz

```

(continues on next page)

(continued from previous page)

```

def rectfn(rect_num, filled):
    """
    We return one rect per page.
    """
    rect = fitz.Rect(10, 20, 290, 380)
    mediabox = fitz.Rect(0, 0, 300, 400)
    #print(f'rectfn(): rect_num={rect_num} filled={filled}')
    return mediabox, rect, None

def contentfn(positions):
    """
    Returns html content, with a table of contents derived from `positions`.
    """
    ret = ''
    ret += textwrap.dedent('''
        <!DOCTYPE html>
        <body>
        <h2>Contents</h2>
        <ul>
        ''')

    # Create table of contents with links to all <h1..6> sections in the
    # document.
    for position in positions:
        if position.heading and (position.open_close & 1):
            text = position.text if position.text else ''
            if position.id:
                ret += f"    <li><a href=\"#{position.id}\">{text}</a>\n"
            else:
                ret += f"    <li>{text}\n"
        ret += f"    <ul>\n"
        ret += f"    <li>page={position.page_num}\n"
        ret += f"    <li>depth={position.depth}\n"
        ret += f"    <li>heading={position.heading}\n"
        ret += f"    <li>id={position.id!r}\n"
        ret += f"    <li>href={position.href!r}\n"
        ret += f"    <li>rect={position.rect}\n"
        ret += f"    <li>text={text!r}\n"
        ret += f"    <li>open_close={position.open_close}\n"
        ret += f"    </ul>\n"

    ret += '</ul>\n'

    # Main content.
    ret += textwrap.dedent(f'''
        <h1>First section</h1>
        <p>Contents of first section.

        <h1>Second section</h1>
        <p>Contents of second section.
    ''')

```

(continues on next page)

(continued from previous page)

```

    <h2>Second section first subsection</h2>

    <p>Contents of second section first subsection.

    <h1>Third section</h1>
    <p>Contents of third section.

    </body>
    '''
    ret = ret.strip()
    with open(__file__.replace('.py', '.html'), 'w') as f:
        f.write(ret)
    return ret;

out_path = __file__.replace('.py', '.pdf')
writer = fitz.DocumentWriter(out_path)
fitz.Story.write_stabilized(writer, contentfn, rectfn)
writer.close()

```

4.10.12.3 How to do iterative layout and create PDF links with `Story.write_stabilized_links()`

This script is similar to the one described in “How to use `Story.write_stabilized()`” above, except that the generated PDF also contains links that correspond to the internal links in the original html.

This is done by using `Story.write_stabilized_links()`; this is slightly different from `Story.write_stabilized()`:

- It does not take a `DocumentWriter` writer arg.
- It returns a PDF `Document` instance.

[The reasons for this are a little involved; for example a `DocumentWriter` is not necessarily a PDF writer, so doesn't really work in a PDF-specific API.]

Files:

- docs/samples/story-write-stabilized-links.py

```

"""
Demo script for PyMuPDF's `fitz.Story.write_stabilized_with_links()`.

`fitz.Story.write_stabilized_links()` is similar to
`fitz.Story.write_stabilized()` except that it creates a PDF `fitz.Document`
that contains PDF links generated from all internal links in the original html.
"""

import textwrap

```

(continues on next page)

(continued from previous page)

```

import fitz

def rectfn(rect_num, filled):
    """
    We return one rect per page.
    """
    rect = fitz.Rect(10, 20, 290, 380)
    mediabox = fitz.Rect(0, 0, 300, 400)
    #print(f'rectfn(): rect_num={rect_num} filled={filled}')
    return mediabox, rect, None

def contentfn(positions):
    """
    Returns html content, with a table of contents derived from `positions`.
    """
    ret = ''
    ret += textwrap.dedent('''
        <!DOCTYPE html>
        <body>
        <h2>Contents</h2>
        <ul>
        ''')

    # Create table of contents with links to all <h1..6> sections in the
    # document.
    for position in positions:
        if position.heading and (position.open_close & 1):
            text = position.text if position.text else ''
            if position.id:
                ret += f"    <li><a href=\"#{position.id}\">{text}</a>\n"
            else:
                ret += f"    <li>{text}\n"
        ret += f"    <ul>\n"
        ret += f"    <li>page={position.page_num}\n"
        ret += f"    <li>depth={position.depth}\n"
        ret += f"    <li>heading={position.heading}\n"
        ret += f"    <li>id={position.id!r}\n"
        ret += f"    <li>href={position.href!r}\n"
        ret += f"    <li>rect={position.rect}\n"
        ret += f"    <li>text={text!r}\n"
        ret += f"    <li>open_close={position.open_close}\n"
        ret += f"    </ul>\n"

    ret += '</ul>\n'

    # Main content.
    ret += textwrap.dedent(f'''
        <h1>First section</h1>
        <p>Contents of first section.
    ''')

```

(continues on next page)

(continued from previous page)

```
<ul>
<li><a href="#idtest">Link to IDTEST</a>.
<li><a href="#nametest">Link to NAMETEST</a>.
</ul>

<h1>Second section</h1>
<p>Contents of second section.
<h2>Second section first subsection</h2>

<p>Contents of second section first subsection.
<p id="idtest">IDTEST

<h1>Third section</h1>
<p>Contents of third section.
<p><a name="nametest">NAMETEST</a>.

</body>
'''
ret = ret.strip()
with open(__file__.replace('.py', '.html'), 'w') as f:
    f.write(ret)
return ret;

out_path = __file__.replace('.py', '.pdf')
document = fitz.Story.write_stabilized_with_links(contentfn, rectfn)
document.save(out_path)
```


MODULE *FITZ*

- New in version 1.16.8

PyMuPDF can also be used in the command line as a **module** to perform utility functions. This feature should obsolete writing some of the most basic scripts.

Admittedly, there is some functional overlap with the MuPDF CLI `mutool`. On the other hand, PDF embedded files are no longer supported by MuPDF, so PyMuPDF is offering something unique here.

5.1 Invocation

Invoke the module like this:

```
python -m fitz <command and parameters>
```

General remarks:

- Request help via "-h", resp. command-specific help via "command -h".
- Parameters may be abbreviated where this does not introduce ambiguities.
- Several commands support parameters `-pages` and `-xrefs`. They are intended for down-selection. Please note that:
 - **page numbers** for this utility must be given **1-based**.
 - valid *xref* numbers start at 1.
 - Specify a comma-separated list of either *single* integers or integer *ranges*. A **range** is a pair of integers separated by one hyphen "-". Integers must not exceed the maximum page, resp. xref number. To specify that maximum, the symbolic variable "N" may be used. Integers or ranges may occur several times, in any sequence and may overlap. If in a range the first number is greater than the second one, the respective items will be processed in reversed order.
- How to use the module inside your script:

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

- Use the following 2-liner and compile it with [Nuitka](#) in standalone mode. This will give you a CLI executable with all the module’s features, that can be used on all compatible platforms without Python, PyMuPDF or MuPDF being installed.

```
from fitz.__main__ import main
main()
```

5.2 Cleaning and Copying

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

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

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

positional arguments:
input                  PDF filename
output                output PDF filename

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

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

Please consult [Document.save\(\)](#) for the parameter meanings.

5.3 Extracting Fonts and Images

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

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

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

positional arguments:
input                  PDF filename

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

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

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

The output directory must already exist.

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

5.4 Joining PDF Documents

To join several PDF files specify:

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

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

positional arguments:
input                  input filenames

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

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

Note:

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

Example: To join the following files

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

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

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

5.5 Low Level Information

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

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

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

positional arguments:
input                  PDF filename

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

Examples:

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

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

python -m fitz show x.pdf -pass jorjmckie
authenticated as owner
file 'x.pdf', pages: 1, objects: 19, 58 MB, PDF 1.4, encryption: Standard V5 R6 256-bit_
```

(continues on next page)

(continued from previous page)

```

↪AES
Document contains 15 embedded files.

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

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

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

```

5.6 Embedded Files Commands

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

5.6.1 Information

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

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

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

positional arguments:
input                  PDF filename

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

Example:

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

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

Detailed output would look like this per entry:

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


5.6.2 Extraction

Extract an embedded file like this:

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

----- extract embedded file to disk -----

positional arguments:
input                  PDF filename

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

For details consult [`Document.embfile_get\(\)`](#). Example (refer to previous section):

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

5.6.3 Deletion

Delete an embedded file like this:

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

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

positional arguments:
input                  PDF filename

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

For details consult [`Document.embfile_del\(\)`](#).

5.6.4 Insertion

Add a new embedded file using this command:

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

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

positional arguments:
input                  PDF filename

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

“*NAME*” **must not** already exist in the PDF. For details consult [*Document.embfile_add\(\)*](#).

5.6.5 Updates

Update an existing embedded file using this command:

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

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

positional arguments:
input                  PDF filename

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

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

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

5.6.6 Copying

Copy embedded files between PDFs:

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

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

positional arguments:
input                  PDF to receive embedded files

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

5.7 Text Extraction

- New in v1.18.16

Extract text from arbitrary supported documents (**not only PDF**) to a textfile. Currently, there are three output formatting modes available: simple, block sorting and reproduction of physical layout.

- **Simple** text extraction reproduces all text as it appears in the document pages – no effort is made to rearrange in any particular reading order.
- **Block sorting** sorts text blocks (as identified by MuPDF) by ascending vertical, then horizontal coordinates. This should be sufficient to establish a “natural” reading order for basic pages of text.
- **Layout** strives to reproduce the original appearance of the input pages. You can expect results like this (produced by the command `python -m fitz gettext -pages 1 demo1.pdf`):

DAIMLER-BENZ

Das Windsor-Syndrom

„Fordern Sie uns! Grillen Sie uns“, appellierte Konzernchef Jürgen Schrepp an seine Führungskräfte. Auf dem Topmanagement-Meeting kontierte er auch die Attacken seines Vorgängers Edzard Reuter.

Wer etwas ist oder sein möchte bei Daimler-Benz, der achtet auf die Kleiderordnung: man trägt Blau im Schwebenkonferenz, hell am Fließband, dunkel auf der Führungsebene. Und wer den Entscheidungssträngen im Vorstand ganz nahe ist, der darf sich OFK-Mitglied nennen, der gehört zum oberen Führungskreis des Hauses.

Ende Januar zogen rund 1000 der 1400 OFK-Mitglieder in die Stuttgarter Liederhalle ein, viele dunkelblau gewandt und alle gespannt wie Chorknaben vor einem großen Auftritt. Jürgen Schrepp (53), der Chef, hatte gerufen, und er verlangte Manesmut: „Dies ist unsere gemeinsame Veranstaltung. Nutzen Sie sie! Fordern Sie uns! Grillen Sie uns!“

Für Spannung bei dem Treffen am 27. Januar war gesorgt: Nie zuvor lagen Markterfolg und Mismanagement so nahe beieinander; nie zuvor rankten sich so viele Gerüchte um Vorstände; nie zuvor hatte ein ehemaliger Vorsitzender so mit dem Unternehmen abgerechnet wie jetzt Edzard Reuter (70) in seinen Memoiren (siehe Kasten Seite 16).

Was sagt Schrepp intern zur peinlichen Elch-Panne, was zum verunglückten Smart? Welche Signale sendet der Vorstandschef in Richtung seiner Kollegen Jürgen Hubbert (58, Pkw-Geschäft) und Dieter Zetsche (44, früher Entwicklung, heute Vertrieb), die für das Debakel die Verantwortung tragen?

Der Mannschaftskapitän ließ sich nicht aus der Reserve locken: „Wir sitzen Ihnen hier als Team gegenüber“, verkündete Schrepp. Ein größeres Revirement, so es denn dazu kommt, bleibt bis nach der Hauptversammlung am 27. Mai tabu.

Dennoch gab es am Rande des Gipfeltreffens zwei Toppersonalisten: Der kantige Horst Zimmer (62), dem es als omnipotentem Chef des Geschäftsbereichs Lkw Europa immer



ziemlich egal war, wer unter ihm den Nutzfahrzeugvorstand abgab, verläßt den Konzern.

Zimmer geht aus gesundheitlichen Gründen im Zenit seiner Karriere: Die beiden neuen Modellreihen Actros (schwere Lastkraftwagen) und Atego (Verteiler-Lkw) sind im

Markt, und seine Division soll 1998 zum erstenmal wieder schwarze Zahlen schreiben. Als Nachfolger wird der erste Mann des Bereichs Lkw-Antriebsstrang, Klaus Maier (44), gehandelt.

Durch rechtzeitige Pensionierung entzieht sich Peter Fietzek (59), Bereichsvorstand und Aktien-Beauftragter, den Folgen der Reorganisation in Fernost.

Künftig regieren in der Region vier „Chief Executives“ (Japan, Indochina, ASEAN, China), die jeweils das gesamte Konzerngeschäft von

Flugmotoren bis Finanzdienstleistungen verantworten.

Auf der Veranstaltung selbst ging es dann zunächst um einen Mann, der seit knapp drei Jahren zu internen Daimler-Foren nicht mehr eingeladen wird: Edzard Reuter. Das Wirken seines Vorgängers kann Schrepp als

DAIMLER-BENZ

Das Windsor-Syndrom

„Fordern Sie uns! Grillen Sie uns“, appellierte Konzernchef Jürgen

Schrepp an seine Führungskräfte. Auf dem Topmanagement-Meeting

kontierte er auch die Attacken seines Vorgängers Edzard Reuter.

er etwas ist oder sein möchte bei Daimler-Benz, der achtet auf die Kleiderordnung: man trägt Blau im Schwebenkonferenz, hell am Fließband, dunkel auf der Führungsebene. Und wer den Entscheidungssträngen im Vorstand ganz nahe ist, der darf sich OFK-Mitglied nennen, der gehört zum oberen Führungskreis des Hauses.

Ende Januar zogen rund 1000 der 1400 OFK-Mitglieder in die Stuttgarter Liederhalle ein, viele dunkelblau gewandt und alle gespannt wie Chorknaben vor einem großen Auftritt. Jürgen Schrepp (53), der Chef, hatte gerufen, und er verlangte Manesmut: „Dies ist unsere gemeinsame Veranstaltung. Nutzen Sie sie! Fordern Sie uns! Grillen Sie uns!“

Für Spannung bei dem Treffen am 27. Januar war gesorgt: Nie zuvor lagen Markterfolg und Mismanagement so nahe beieinander; nie zuvor rankten sich so viele Gerüchte um Vorstände; nie zuvor hatte ein ehemaliger Vorsitzender so mit dem Unternehmen abgerechnet wie jetzt Edzard Reuter (70) in seinen Memoiren (siehe Kasten Seite 16).

Was sagt Schrepp intern zur peinlichen Elch-Panne, was zum verunglückten Smart? Welche Signale sendet der Vorstandschef in Richtung seiner Kollegen Jürgen Hubbert (58, Pkw-Geschäft) und Dieter Zetsche (44, früher Entwicklung, heute Vertrieb), die für das Debakel die Verantwortung tragen?

Der Mannschaftskapitän ließ sich nicht aus der Reserve locken: „Wir sitzen Ihnen hier als Team gegenüber“, verkündete Schrepp. Ein größeres Revirement, so es denn dazu kommt, bleibt bis nach der Hauptversammlung am 27. Mai tabu.

Markt, und seine Division soll 1998 zum erstenmal wieder schwarze Zahlen schreiben. Als Nachfolger wird der erste Mann des Bereichs Lkw-Antriebsstrang, Klaus Maier (44), gehandelt.

Durch rechtzeitige Pensionierung entzieht sich Peter Fietzek (59), Bereichsvorstand und Aktien-Beauftragter, den Folgen der Reorganisation in Fernost.

Künftig regieren in der Region vier „Chief Executives“ (Japan, Indochina, ASEAN, China), die jeweils das gesamte Konzerngeschäft von

Flugmotoren bis Finanzdienstleistungen verantworten.

Auf der Veranstaltung selbst ging es dann zunächst um einen Mann, der seit knapp drei Jahren zu internen Daimler-Foren nicht mehr eingeladen wird: Edzard Reuter. Das Wirken seines Vorgängers kann Schrepp als

Note: The “gettext” command offers a functionality similar to the CLI tool pdftotext by XPDF software, <http://www.foolabs.com/xpdf/> – this is especially true for “layout” mode, which combines that tool’s -layout and -table options.

After each page of the output file, a formfeed character, hex(12) is written – even if the input page has no text at all. This behavior can be controlled via options.

Note: For “layout” mode, **only horizontal, left-to-right, top-to bottom** text is supported, other text is ignored. In this mode, text is also ignored, if its fontsize is too small.

“Simple” and “blocks” mode in contrast output **all text** for any text size or orientation.

Command:

```
python -m fitz gettext -h
usage: fitz gettext [-h] [-password PASSWORD] [-mode {simple,blocks,layout}] [-pages_
↵PAGES] [-noligatures]
                        [-convert-white] [-extra-spaces] [-noformfeed] [-skip-empty] [-
↵output OUTPUT] [-grid GRID]
                        [-fontsize FONTSIZE]
                        input
```

----- extract text in various formatting modes -----

positional arguments:

input input document filename

optional arguments:

(continues on next page)

(continued from previous page)

```

-h, --help          show this help message and exit
-password PASSWORD  password for input document
-mode {simple,blocks,layout}
                    mode: simple, block sort, or layout (default)
-pages PAGES        select pages, format: 1,5-7,50-N
-noligatures        expand ligature characters (default False)
-convert-white      convert whitespace characters to space (default False)
-extra-spaces       fill gaps with spaces (default False)
-noformfeed         write linefeeds, no formfeeds (default False)
-skip-empty         suppress pages with no text (default False)
-output OUTPUT      store text in this file (default inputfilename.txt)
-grid GRID          merge lines if closer than this (default 2)
-fontsize FONTSIZE  only include text with a larger fontsize (default 3)

```

Note: Command options may be abbreviated as long as no ambiguities are introduced. So the following do the same:

- ... -output text.txt -noligatures -noformfeed -convert-white -grid 3 -extra-spaces ...
- ... -o text.txt -nol -nof -c -g 3 -e ...

The output filename defaults to the input with its extension replaced by `.txt`. As with other commands, you can select page ranges (**caution: 1-based!**) in mutool format, as indicated above.

- **mode:** (str) select a formatting mode – default is “layout”.
- **noligatures:** (bool) corresponds to **not** `TEXT_PRESERVE_LIGATURES`. If specified, ligatures (present in advanced fonts: glyphs combining multiple characters like “fi”) are split up into their components (i.e. “f”, “i”). Default is passing them through.
- **convert-white:** corresponds to **not** `TEXT_PRESERVE_WHITESPACE`. If specified, all white space characters (like tabs) are replaced with one or more spaces. Default is passing them through.
- **extra-spaces:** (bool) corresponds to **not** `TEXT_INHIBIT_SPACES`. If specified, large gaps between adjacent characters will be filled with one or more spaces. Default is off.
- **noformfeed:** (bool) instead of `hex(12)` (formfeed), write linebreaks `\n` at end of output pages.
- **skip-empty:** (bool) skip pages with no text.
- **grid:** lines with a vertical coordinate difference of no more than this value (in points) will be merged into the same output line. Only relevant for “layout” mode. **Use with care:** 3 or the default 2 should be adequate in most cases. If **too large**, lines that are *intended* to be different in the original may be merged and will result in garbled and / or incomplete output. If **too low**, artifact separate output lines may be generated for some spans in the input line, just because they are coded in a different font with slightly deviating properties.
- **fontsize:** include text with fontsize larger than this value only (default 3). Only relevant for “layout” option.

CLASSES

6.1 Annot

This class is supported for PDF documents only.

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

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

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

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

Attribute	Short Description
<code>Annot.delete_responses()</code>	delete all responding annotations
<code>Annot.get_file()</code>	get attached file content
<code>Annot.get_oc()</code>	get <i>xref</i> of an <i>OCG</i> / <i>OCMD</i>
<code>Annot.get_pixmap()</code>	image of the annotation as a pixmap
<code>Annot.get_sound()</code>	get the sound of an audio annotation
<code>Annot.get_text()</code>	extract annotation text
<code>Annot.get_textbox()</code>	extract annotation text
<code>Annot.set_border()</code>	set annotation’s border properties
<code>Annot.set_blendmode()</code>	set annotation’s blend mode
<code>Annot.set_colors()</code>	set annotation’s colors
<code>Annot.set_flags()</code>	set annotation’s flags field
<code>Annot.set_irt_xref()</code>	define the annotation to being “In Response To”
<code>Annot.set_name()</code>	set annotation’s name field
<code>Annot.set_oc()</code>	set <i>xref</i> to an <i>OCG</i> / <i>OCMD</i>
<code>Annot.set_opacity()</code>	change transparency
<code>Annot.set_open()</code>	open / close annotation or its Popup
<code>Annot.set_popup()</code>	create a Popup for the annotation

continues on next page

Table 1 – continued from previous page

Attribute	Short Description
<code>Annot.set_rect()</code>	change annotation rectangle
<code>Annot.set_rotation()</code>	change rotation
<code>Annot.update_file()</code>	update attached file content
<code>Annot.update()</code>	apply accumulated annot changes
<code>Annot.blendmode</code>	annotation BlendMode
<code>Annot.border</code>	border details
<code>Annot.colors</code>	border / background and fill colors
<code>Annot.file_info</code>	get attached file information
<code>Annot.flags</code>	annotation flags
<code>Annot.has_popup</code>	whether annotation has a Popup
<code>Annot.irt_xref</code>	annotation to which this one responds
<code>Annot.info</code>	various information
<code>Annot.is_open</code>	whether annotation or its Popup is open
<code>Annot.line_ends</code>	start / end appearance of line-type annotations
<code>Annot.next</code>	link to the next annotation
<code>Annot.opacity</code>	the annot's transparency
<code>Annot.parent</code>	page object of the annotation
<code>Annot.popup_rect</code>	rectangle of the annotation's Popup
<code>Annot.popup_xref</code>	the PDF <i>xref</i> number of the annotation's Popup
<code>Annot.rect</code>	rectangle containing the annotation
<code>Annot.type</code>	type of the annotation
<code>Annot.vertices</code>	point coordinates of Polygons, PolyLines, etc.
<code>Annot.xref</code>	the PDF <i>xref</i> number

Class API

class Annot

get_pixmap(*matrix*=[`fitz.Identity`](#), *dpi*=*None*, *colorspace*=[`fitz.csRGB`](#), *alpha*=*False*)

- Changed in v1.19.2: added support of *dpi* parameter.

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). **All parameters are keyword only.**

Parameters

- **matrix** (*matrix_like*) – a matrix to be used for image creation. Default is [*Identity*](#).
- **dpi** (*int*) – (new in v1.19.2) desired resolution in dots per inch. If not *None*, the matrix parameter is ignored.
- **colorspace** (*Colorspace*) – a colorspace to be used for image creation. Default is [*fitz.csRGB*](#).
- **alpha** (*bool*) – whether to include transparency information. Default is *False*.

Return type

[*Pixmap*](#)

Note: If the annotation has just been created or modified, you should reload the page first via [`page = doc.reload_page\(page\)`](#).

get_text(*opt*, *clip*=None, *flags*=None)

- New in 1.18.0

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

Parameters

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

get_textbox(*rect*)

- New in 1.18.0

Return the annotation text. Mostly (except line breaks) equal to [Annot.get_text\(\)](#) with the “text” option.

Parameters

rect (*rect-like*) – the area to consider, defaults to [Annot.rect](#).

set_info(*info*=None, *content*=None, *title*=None, *creationDate*=None, *modDate*=None, *subject*=None)

- Changed in version 1.16.10

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

Parameters

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

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

set_line_ends(*start*, *end*)

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

Note:

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

Parameters

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

set_oc(*xref*)

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

Parameters

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

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

get_oc()

Return the [xref](#) of an optional content object, or zero if there is none.

Returns

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

set_irt_xref(*xref*)

- New in v1.19.3

Set annotation to be "In Response To" another one.

Parameters

xref (*int*) – The [xref](#) of another annotation.

Note: Must refer to an existing annotation on this page. Setting this property requires no subsequent [update\(\)](#).

set_open(value)

- New in v1.18.4

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

Parameters

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

set_popup(rect)

- New in v1.18.4

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

Parameters

rect (*rect_like*) – the desired rectangle.

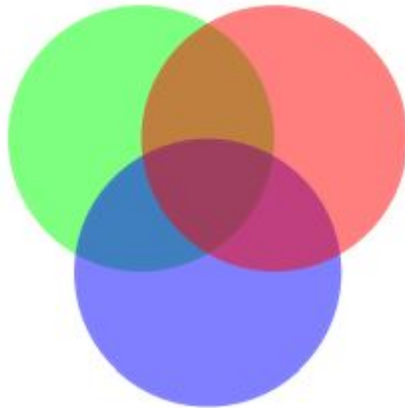
set_opacity(value)

Set the annotation's transparency. Opacity can also be set in [Annot.update\(\)](#).

Parameters

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

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

**blendmode**

- New in v1.18.4

The annotation's blend mode. See [Adobe PDF References](#), page 324 for explanations.

Return type

str

Returns

the blend mode or *None*.

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

set_blendmode(blendmode)

- New in v1.16.14

Set the annotation's blend mode. See *Adobe PDF References*, page 324 for explanations. The blend mode can also be set in *Annot.update()*.

Parameters

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

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

set_name(name)

- New in version 1.16.0

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

Parameters

name (str) – the new name.

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

set_rect(rect)

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

Parameters

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

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

set_rotation(angle)

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

Parameters

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

Note:

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

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

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

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 be set to “D” (dashed).
- **width** (*float*) – see above.
- **style** (*str*) – see above.
- **dashes** (*sequence*) – see above.

set_flags(*flags*)

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

Parameters

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

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

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

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

Parameters

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

Changed in v1.18.5: To completely remove a color specification, use an empty sequence like []. If you specify None, an existing specification will not be changed.

delete_responses()

- New in version 1.16.12

Delete annotations referring to this one. This includes any ‘Popup’ annotations and all annotations responding to it.

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

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

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

- [`Annot.set_rect\(\)`](#)
- [`Annot.set_flags\(\)`](#)
- [`Annot.set_oc\(\)`](#)
- [`Annot.update_file\(\)`](#)
- [`Annot.set_info\(\)`](#) (except any changes to “content”)

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

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

Parameters

- **opacity** (*float*) – (*new in v1.16.14*) **valid for all annotation types**: change or set the annotation’s transparency. Valid values are $0 \leq \text{opacity} < 1$.
- **blend_mode** (*str*) – (*new in v1.16.14*) **valid for all annotation types**: change or set the annotation’s blend mode. For valid values see [*PDF Standard Blend Modes*](#).
- **fontsize** (*float*) – change font size of the text. ‘FreeText’ annotations only.
- **text_color** (*sequence, float*) – change the text color. ‘FreeText’ annotations only.
- **border_color** (*sequence, float*) – change the border color. ‘FreeText’ annotations only.
- **fill_color** (*sequence, float*) – the fill color.
 - ‘Line’, ‘Polyline’, ‘Polygon’ annotations: use it to give applicable line end symbols a fill color other than that of the annotation (*changed in v1.16.16*).
- **cross_out** (*bool*) – (*new in v1.17.2*) add two diagonal lines to the annotation rectangle. ‘Redact’ annotations only. If not desired, *False* must be specified even if the annotation was created with *False*.
- **rotate** (*int*) – new rotation value. Default (-1) means no change. Supports ‘FreeText’ and several other annotation types (see [`Annot.set_rotation\(\)`](#)),¹. Only choose 0, 90, 180, or 270 degrees for ‘FreeText’. Otherwise any integer is acceptable.

Return type

bool

Note: Using this method inside a [`Page.annots\(\)`](#) loop is **not recommended!** This is because most annotation updates require the owning page to be reloaded – which cannot be done inside this loop. Please use the example coding pattern given in the documentation of this generator.

¹ Rotating an annotation also changes its rectangle. Depending on how the annotation was defined, the original rectangle is **not reconstructible** by setting the rotation value to zero again and will be lost.

file_info

Basic information of the annot's attached file.

Return type

dict

Returns

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

get_file()

Returns attached file content.

Return type

bytes

Returns

the content of the attached file.

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

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

Parameters

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

get_sound()

Return the embedded sound of an audio annotation.

Return type

dict

Returns

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

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

opacity

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

Return type

float

parent

The owning page object of the annotation.

Return type

Page

rotation

The annot rotation.

Return type

int

Returns

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

rect

The rectangle containing the annotation.

Return type

Rect

next

The next annotation on this page or None.

Return type

Annot

type

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

Return type

list

info

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

- *name* – e.g. for 'Stamp' annotations it will contain the stamp text like "Sold" or "Experimental", for other annot types you will see the name of the annot's icon here ("PushPin" for FileAttachment).
- *content* – a string containing the text for type *Text* and *FreeText* annotations. Commonly used for filling the text field of annotation pop-up windows.
- *title* – a string containing the title of the annotation pop-up window. By convention, this is used for the **annotation author**.
- *creationDate* – creation timestamp.
- *modDate* – last modified timestamp.
- *subject* – subject.
- *id* – (new in version 1.16.10) a unique identification of the annotation. This is taken from PDF key */NM*. Annotations added by PyMuPDF will have a unique name, which appears here.

Return type

dict

flags

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

Return type

int

line_ends

A pair of integers specifying start and end symbol of annotations types ‘FreeText’, ‘Line’, ‘PolyLine’, and ‘Polygon’. *None* if not applicable. For possible values and descriptions in this list, see the [Adobe PDF References](#), table 1.76 on page 400.

Return type

tuple

vertices

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

- ‘Line’ – the starting and ending coordinates (2 float pairs).
- ‘FreeText’ – 2 or 3 float pairs designating the starting, the (optional) knee point, and the ending coordinates.
- ‘PolyLine’ / ‘Polygon’ – the coordinates of the edges connected by line pieces (n float pairs for n points).
- text markup annotations – 4 float pairs specifying the *QuadPoints* of the marked text span (see [Adobe PDF References](#), page 403).
- ‘Ink’ – list of one to many sublists of vertex coordinates. Each such sublist represents a separate line in the drawing.

Return type

list

colors

dictionary of two lists of floats in range $0 \leq float \leq 1$ specifying the “stroke” and the interior (“fill”) colors. The stroke color is used for borders and everything that is actively painted or written (“stroked”). The fill color is used for the interior of objects like line ends, circles and squares. The lengths of these lists implicitly determine the colorspace used: 1 = GRAY, 3 = RGB, 4 = CMYK. So “[1.0, 0.0, 0.0]” stands for RGB color red. Both lists can be empty if no color is specified.

Return type

dict

xref

The PDF [xref](#).

Return type

int

irt_xref

The PDF [xref](#) of an annotation to which this one responds. Return zero if this is no response annotation.

Return type

int

popup_xref

The PDF [xref](#) of the associated Popup annotation. Zero if non-existent.

Return type

int

has_popup

Whether the annotation has a Popup annotation.

Return type

bool

is_open

Whether the annotation's Popup is open – **or** the annotation itself ('Text' annotations only).

Return type

bool

popup_rect

The rectangle of the associated Popup annotation. Infinite rectangle if non-existent.

Return type*Rect***border**

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

- *width* – a float indicating the border thickness in points. The value is -1.0 if no width is specified.
- *dashes* – a sequence of integers specifying a line dash pattern. *[]* means no dashes, *[n]* means equal on-off lengths of *n* points, longer lists will be interpreted as specifying alternating on-off length values. See the *Adobe PDF References* page 126 for more details.
- *style* – 1-byte border style: “**S**” (Solid) = solid rectangle surrounding the annotation, “**D**” (Dashed) = dashed rectangle surrounding the annotation, the dash pattern is specified by the *dashes* entry, “**B**” (Beveled) = a simulated embossed rectangle that appears to be raised above the surface of the page, “**I**” (Inset) = a simulated engraved rectangle that appears to be recessed below the surface of the page, “**U**” (Underline) = a single line along the bottom of the annotation rectangle.

Return type

dict

6.1.1 Annotation Icons in MuPDF

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

	PushPin
	Graph
	Paperclip
	Tag
	Note
	Comment
	Help
	Insert
	Key
	NewParagraph
	Paragraph
	Mic
	Speaker
	Star (default if none of the above)

6.1.2 Example

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

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

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

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

(continues on next page)

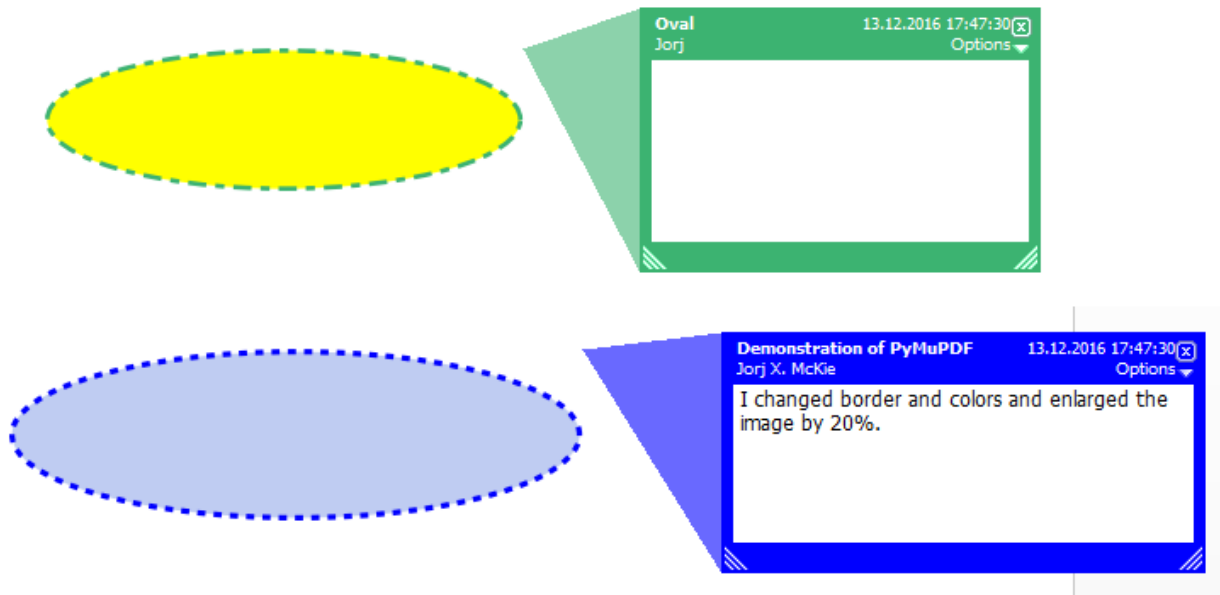
(continued from previous page)

```

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

```

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



6.2 Archive

- New in v1.21.0

This class represents a generalization of file folders and container files like ZIP and TAR archives. Archives allow accessing arbitrary collections of file folders, ZIP / TAR files and single binary data elements as if they all were part of one hierarchical tree of folders.

In PyMuPDF, archives are currently only used by *Story* objects to specify where to look for fonts, images and other resources.

Method / Attribute	Short Description
<code>Archive.add()</code>	add new data to the archive
<code>Archive.has_entry()</code>	check if given name is a member
<code>Archive.read_entry()</code>	read the data given by the name
<code>Archive.entry_list</code>	list[dict] of archive items

Class API

class Archive

```
__init__(self[, content[, path]])
```

Creates a new archive. Without parameters, an empty archive is created.

If provided, `content` may be one of the following:

- another Archive: the archive is being made a sub-archive of the new one.
- a string: this must be the name of a local folder or file. `pathlib.Path` objects are also supported.
 - A **folder** will be converted to a sub-archive, so its files (and any sub-folders) can be accessed by their names.
 - A **file** will be read with mode "rb" and these binary data (a `bytes` object) be treated as a single-member sub-archive. In this case, the `path` parameter is **mandatory** and should be the member name under which this item can be found / retrieved.
- a `zipfile.ZipFile` or `tarfile.TarFile` object: Will be added as a sub-archive.
- a Python binary object (`bytes`, `bytearray`, `io.BytesIO`): this will add a single-member sub-archive. In this case, the `path` parameter is **mandatory** and should be the member name under which this item can be found / retrieved.
- a tuple (`data`, `name`): This will add a single-member sub-archive with the member name `name`. `data` may be a Python binary object or a local file name (in which case its binary file content is used). Use this format if you need to specify `path`.
- a Python sequence: This is a convenience format to specify any combination of the above.

If provided, `path` must be a string.

- If `content` is either binary data or a file name, this parameter is mandatory and must be the name under which the data can be found.
- Otherwise this parameter is optional. It can be used to simulate a folder name or a mount point, under which this sub-archive's elements can be found. For example this specification `Archive((data, "name"), "path")` means that `data` will be found using the element name `"path/name"`. Similar is true for other sub-archives: to retrieve members of a ZIP sub-archive, their names must be prefixed with `"path/"`. The main purpose of this parameter probably is to differentiate between duplicate names.

Note: If duplicate entry names exist in the archive, always the last entry with that name will be found / retrieved. During archive creation, or appending more data to an archive (see [Archive.add\(\)](#)) no check for duplicates will be made. Use the `path` parameter to prevent this from happening.

add(`content`[, `path`])

Append a sub-archive. The meaning of the parameters are exactly the same as explained above. Of course, parameter `content` is not optional here.

has_entry(`name`)

Checks whether an entry exists in any of the sub-archives.

Parameters

name (`str`) – The fully qualified name of the entry. So must include any path prefix under which the entry's sub-archive has been added.

Returns

True or False.

read_entry(`name`)

Retrieve the data of an entry.

Parameters

name (`str`) – The fully qualified name of the entry. So must include any path prefix under which the entry's sub-archive has been added.

Returns

The binary data (bytes) of the entry. If not found, an exception is raised.

entry_list

A list of the archive's sub-archives. Each list item is a dictionary with the following keys:

- **entries** – a list of (top-level) entry names in this sub-archive.
- **fmt** – the format of the sub-archive. This is one of the strings “dir” (file folder), “zip” (ZIP archive), “tar” (TAR archive), or “tree” for single binary entries or file content.
- **path** – the value of the *path* parameter under which this sub-archive was added.

Example:

```
>>> from pprint import pprint
>>> import fitz
>>> dir1 = "fitz-32" # a folder name
>>> dir2 = "fitz-64" # a folder name
>>> img = ("nur-ruhig.jpg", "img") # an image file
>>> members = (dir1, img, dir2) # we want to append these in one go
>>> arch = fitz.Archive()
>>> arch.add(members, path="mypath")
>>> pprint(arch.entry_list)
[{'entries': ['310', '37', '38', '39'], 'fmt': 'dir', 'path': 'mypath'},
 {'entries': ['img'], 'fmt': 'tree', 'path': 'mypath'},
 {'entries': ['310', '311', '37', '38', '39', 'pypy'],
  'fmt': 'dir',
  'path': 'mypath'}]
>>>
```

6.3 Colorspace

Represents the color space of a *Pixmap*.

Class API**class Colorspace**

__init__(self, n)

Constructor

Parameters

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

name

The name identifying the colorspace. Example: *fitz.csCMYK.name* = ‘DeviceCMYK’.

Type

str

n

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

type
int

Predefined Colorspaces

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

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

6.4 DisplayList

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

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

A display list is populated with objects from a page, usually by executing `Page.get_displaylist()`. There also exists an independent constructor.

“Replay” the list (once or many times) by invoking one of its methods `run()`, `get_pixmap()` or `get_textpage()`.

Method	Short Description
<code>run()</code>	Run a display list through a device.
<code>get_pixmap()</code>	generate a pixmap
<code>get_textpage()</code>	generate a text page
<code>rect</code>	mediabox of the display list

Class API

class DisplayList

__init__(*self*, *mediabox*)

Create a new display list.

Parameters

mediabox (*Rect*) – The page’s rectangle.

Return type

DisplayList

run(*device*, *matrix*, *area*)

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

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

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.

get_pixmap(*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_like*) – restrict rendering to the intersection of this area with *DisplayList.rect*.

Return type

Pixmap

Returns

pixmap of the display list.

get_textpage(*flags*)

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

Parameters

flags (*int*) – control which information is parsed into a text page. Default value in PyMuPDF is 3 = `TEXT_PRESERVE_LIGATURES` | `TEXT_PRESERVE_WHITESPACE`, i.e. ligatures are **passed through**, white spaces are **passed through** (not translated to spaces), and images are **not included**. See *Text Extraction 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.get_displaylist()*.

Type

Rect

6.5 Document

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

There exists the alias *open* for this class, i.e. `fitz.Document(...)` and `fitz.open(...)` do exactly the same thing.

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

Note: Starting with v1.17.0, a new page addressing mechanism for **EPUB files only** is supported. This document type is internally organized in chapters such that pages can most efficiently be found by their so-called “location”. The

location is a tuple (*chapter*, *pno*) consisting of the chapter number and the page number **in that chapter**. Both numbers are zero-based.

While it is still possible to locate a page via its (absolute) number, doing so may mean that the complete EPUB document must be layouted before the page can be addressed. This may have a significant performance impact if the document is very large. Using the page's (*chapter*, *pno*) prevents this from happening.

To maintain a consistent API, PyMuPDF supports the page *location* syntax for **all file types** – documents without this feature simply have just one chapter. `Document.load_page()` and the equivalent index access now also support a *location* argument.

There are a number of methods for converting between page numbers and locations, for determining the chapter count, the page count per chapter, for computing the next and the previous locations, and the last page location of a document.

Method / Attribute	Short Description
<code>Document.add_layer()</code>	PDF only: make new optional content configuration
<code>Document.add_ocg()</code>	PDF only: add new optional content group
<code>Document.authenticate()</code>	gain access to an encrypted document
<code>Document.can_save_incrementally()</code>	check if incremental save is possible
<code>Document.chapter_page_count()</code>	number of pages in chapter
<code>Document.close()</code>	close the document
<code>Document.convert_to_pdf()</code>	write a PDF version to memory
<code>Document.copy_page()</code>	PDF only: copy a page reference
<code>Document.del_toc_item()</code>	PDF only: remove a single TOC item
<code>Document.delete_page()</code>	PDF only: delete a page
<code>Document.delete_pages()</code>	PDF only: delete multiple pages
<code>Document.embfile_add()</code>	PDF only: add a new embedded file from buffer
<code>Document.embfile_count()</code>	PDF only: number of embedded files
<code>Document.embfile_del()</code>	PDF only: delete an embedded file entry
<code>Document.embfile_get()</code>	PDF only: extract an embedded file buffer
<code>Document.embfile_info()</code>	PDF only: metadata of an embedded file
<code>Document.embfile_names()</code>	PDF only: list of embedded files
<code>Document.embfile_upd()</code>	PDF only: change an embedded file
<code>Document.extract_font()</code>	PDF only: extract a font by <i>xref</i>
<code>Document.extract_image()</code>	PDF only: extract an embedded image by <i>xref</i>
<code>Document.ez_save()</code>	PDF only: <code>Document.save()</code> with different defaults
<code>Document.find_bookmark()</code>	retrieve page location after layouting document
<code>Document.fullcopy_page()</code>	PDF only: duplicate a page
<code>Document.get_layer()</code>	PDF only: lists of OCGs in ON, OFF, RBGroups
<code>Document.get_layers()</code>	PDF only: list of optional content configurations
<code>Document.get_oc()</code>	PDF only: get OCG /OCMD xref of image / form xobject
<code>Document.get_ocgs()</code>	PDF only: info on all optional content groups
<code>Document.get_ocmd()</code>	PDF only: retrieve definition of an <i>OCMD</i>
<code>Document.get_page_fonts()</code>	PDF only: list of fonts referenced by a page
<code>Document.get_page_images()</code>	PDF only: list of images referenced by a page
<code>Document.get_page_labels()</code>	PDF only: list of page label definitions
<code>Document.get_page_numbers()</code>	PDF only: get page numbers having a given label
<code>Document.get_page_pixmap()</code>	create a pixmap of a page by page number
<code>Document.get_page_text()</code>	extract the text of a page by page number
<code>Document.get_page_xobjects()</code>	PDF only: list of XObjects referenced by a page
<code>Document.get_sigflags()</code>	PDF only: determine signature state
<code>Document.get_toc()</code>	extract the table of contents

continues on next page

Table 2 – continued from previous page

Method / Attribute	Short Description
<code>Document.get_xml_metadata()</code>	PDF only: read the XML metadata
<code>Document.has_annots()</code>	PDF only: check if PDF contains any annots
<code>Document.has_links()</code>	PDF only: check if PDF contains any links
<code>Document.insert_page()</code>	PDF only: insert a new page
<code>Document.insert_pdf()</code>	PDF only: insert pages from another PDF
<code>Document.journal_can_do()</code>	PDF only: which journal actions are possible
<code>Document.journal_enable()</code>	PDF only: enables journalling for the document
<code>Document.journal_load()</code>	PDF only: load journal from a file
<code>Document.journal_op_name()</code>	PDF only: return name of a journalling step
<code>Document.journal_position()</code>	PDF only: return journalling status
<code>Document.journal_redo()</code>	PDF only: redo current operation
<code>Document.journal_save()</code>	PDF only: save journal to a file
<code>Document.journal_start_op()</code>	PDF only: start an “operation” giving it a name
<code>Document.journal_stop_op()</code>	PDF only: end current operation
<code>Document.journal_undo()</code>	PDF only: undo current operation
<code>Document.layer_ui_configs()</code>	PDF only: list of optional content intents
<code>Document.layout()</code>	re-paginate the document (if supported)
<code>Document.load_page()</code>	read a page
<code>Document.make_bookmark()</code>	create a page pointer in reflowable documents
<code>Document.move_page()</code>	PDF only: move a page to different location in doc
<code>Document.need_appearances()</code>	PDF only: get/set <code>/NeedAppearances</code> property
<code>Document.new_page()</code>	PDF only: insert a new empty page
<code>Document.next_location()</code>	return (chapter, pno) of following page
<code>Document.outline_xref()</code>	PDF only: xref a TOC item
<code>Document.page_cropbox()</code>	PDF only: the unrotated page rectangle
<code>Document.page_xref()</code>	PDF only: xref of a page number
<code>Document.pages()</code>	iterator over a page range
<code>Document.pdf_catalog()</code>	PDF only: xref of catalog (root)
<code>Document.pdf_trailer()</code>	PDF only: trailer source
<code>Document.prev_location()</code>	return (chapter, pno) of preceeding page
<code>Document.reload_page()</code>	PDF only: provide a new copy of a page
<code>Document.save()</code>	PDF only: save the document
<code>Document.saveIncr()</code>	PDF only: save the document incrementally
<code>Document.scrub()</code>	PDF only: remove sensitive data
<code>Document.search_page_for()</code>	search for a string on a page
<code>Document.select()</code>	PDF only: select a subset of pages
<code>Document.set_layer_ui_config()</code>	PDF only: set OCG visibility temporarily
<code>Document.set_layer()</code>	PDF only: mass changing OCG states
<code>Document.set_metadata()</code>	PDF only: set the metadata
<code>Document.set_oc()</code>	PDF only: attach OCG/OCMD to image / form xobject
<code>Document.set_ocmd()</code>	PDF only: create or update an OCMD
<code>Document.set_page_labels()</code>	PDF only: add/update page label definitions
<code>Document.set_toc_item()</code>	PDF only: change a single TOC item
<code>Document.set_toc()</code>	PDF only: set the table of contents (TOC)
<code>Document.set_xml_metadata()</code>	PDF only: create or update document XML metadata
<code>Document.subset_fonts()</code>	PDF only: create font subsets
<code>Document.switch_layer()</code>	PDF only: activate OC configuration
<code>Document.tobytes()</code>	PDF only: writes document to memory
<code>Document.xref_copy()</code>	PDF only: copy a PDF dictionary to another xref

continues on next page

Table 2 – continued from previous page

Method / Attribute	Short Description
<code>Document.xref_get_key()</code>	PDF only: get the value of a dictionary key
<code>Document.xref_get_keys()</code>	PDF only: list the keys of object at <i>xref</i>
<code>Document.xref_object()</code>	PDF only: get the definition source of <i>xref</i>
<code>Document.xref_set_key()</code>	PDF only: set the value of a dictionary key
<code>Document.xref_stream_raw()</code>	PDF only: raw stream source at <i>xref</i>
<code>Document.xref_xml_metadata()</code>	PDF only: <i>xref</i> of XML metadata
<code>Document.chapter_count</code>	number of chapters
<code>Document.FormFonts</code>	PDF only: list of global widget fonts
<code>Document.is_closed</code>	has document been closed?
<code>Document.is_dirty</code>	PDF only: has document been changed yet?
<code>Document.is_encrypted</code>	document (still) encrypted?
<code>Document.is_form_pdf</code>	is this a Form PDF?
<code>Document.is_pdf</code>	is this a PDF?
<code>Document.is_reflowable</code>	is this a reflowable document?
<code>Document.is_repaired</code>	PDF only: has this PDF been repaired during open?
<code>Document.last_location</code>	(chapter, pno) of last page
<code>Document.metadata</code>	metadata
<code>Document.name</code>	filename of document
<code>Document.needs_pass</code>	require password to access data?
<code>Document.outline</code>	first <i>Outline</i> item
<code>Document.page_count</code>	number of pages
<code>Document.permissions</code>	permissions to access the document

Class API

class Document

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

- Changed in v1.14.13: support `io.BytesIO` for memory documents.
- Changed in v1.19.6: Clearer, shorter and more consistent exception messages. File type “pdf” is always assumed if not specified. Empty files and memory areas will always lead to exceptions.

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, if not a PDF, either *filename* or *filetype* must indicate its type.
- If *stream* is *None*, then a document is created from the file given by *filename*. Its type is inferred from the extension. This can be overruled by *filetype*.

Parameters

- **filename** (*str*, *pathlib*) – A UTF-8 string or *pathlib* object containing a file path. The document type is inferred from the filename extension. If not present or not matching a supported type, a PDF document is assumed. For memory documents, this argument may be used instead of *filetype*, see below.
- **stream** (*bytes*, *bytearray*, *BytesIO*) – A memory area containing a supported document. If not a PDF, its type **must** be specified by either *filename* or *filetype*.
- **filetype** (*str*) – A string specifying the type of document. This may be anything 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” or “.pdf” will also work. May be omitted for PDF documents, otherwise

must match a supported document type.

- **rect** (*rect_like*) – a rectangle specifying the desired page size. This parameter is only meaningful for documents with a variable page layout (“reflowable” documents), like e-books or HTML, and ignored otherwise. If specified, it must be a non-empty, finite rectangle with top-left coordinates (0, 0). Together with parameter *fontsize*, each page will be accordingly laid out and hence also determine the number of pages.
- **width** (*float*) – may used together with *height* as an alternative to *rect* to specify layout information.
- **height** (*float*) – may used together with *width* as an alternative to *rect* to specify layout information.
- **fontsize** (*float*) – the default fontsize for reflowable document types. This parameter is ignored if none of the parameters *rect* or *width* and *height* are specified. Will be used to calculate the page layout.

Raises

- **TypeError** – if the *type* of any parameter does not conform.
- **FileNotFoundError** – if the file / path cannot be found. Re-implemented as subclass of `RuntimeError`.
- **EmptyFileError** – if the file / path is empty or the bytes object in memory has zero length. A subclass of `FileDataError` and `RuntimeError`.
- **ValueError** – if an unknown file type is explicitly specified.
- **FileDataError** – if the document has an invalid structure for the given type – or is no file at all (but e.g. a folder). A subclass of `RuntimeError`.

Returns

A document object. If the document cannot be created, an exception is raised in the above sequence. Note that PyMuPDF-specific exceptions, `FileNotFoundError`, `EmptyFileError` and `FileDataError` are intercepted if you check for `RuntimeError`.

In case of problems you can see more detail in the internal messages store: `print(fitz.TOOLS.mupdf_warnings())` (which will be emptied by this call, but you can also prevent this – consult [Tools.mupdf_warnings\(\)](#)).

Note: Not all document types are checked for valid formats already at open time. Raster images for example will raise exceptions only later, when trying to access the content. Other types (notably with non-binary content) may also be opened (and sometimes **accessed**) successfully – sometimes even when having invalid content for the format:

- HTM, HTML, XHTML: **always** opened, `metadata["format"]` is “HTML5”, resp. “XHTML”.
 - XML, FB2: **always** opened, `metadata["format"]` is “FictionBook2”.
-

Overview of possible forms, note: `open` is a synonym of `Document`:

```
>>> # from a file
>>> doc = fitz.open("some.xps")
>>> # handle wrong extension
>>> doc = fitz.open("some.file", filetype="xps")
>>>
>>> # from memory, filetype is required if not a PDF
>>> doc = fitz.open("xps", mem_area)
>>> doc = fitz.open(None, mem_area, "xps")
>>> doc = fitz.open(stream=mem_area, filetype="xps")
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # new empty PDF
>>> doc = fitz.open()
>>> doc = fitz.open(None)
>>> doc = fitz.open("")
```

Note: Raster images with a wrong (but supported) file extension **are no problem**. MuPDF will determine the correct image type when file **content** is actually accessed and will process it without complaint. So `fitz.open("file.jpg")` will work even for a PNG image.

The Document class can also be used as a **context manager**. On exit, the document will automatically be closed.

```
>>> import fitz
>>> with fitz.open(...) as doc:
    for page in doc: print("page %i" % page.number)
page 0
page 1
page 2
page 3
>>> doc.is_closed
True
>>>
```

`get_oc(xref)`

- New in v1.18.4

Return the cross reference number of an *OCG* or *OCMD* attached to an image or form xobject.

Parameters

xref (*int*) – the *xref* of an image or form xobject. Valid such cross reference numbers are returned by `Document.get_page_images()`, resp. `Document.get_page_xobjects()`. For invalid numbers, an exception is raised.

Return type

int

Returns

the cross reference number of an optional contents object or zero if there is none.

`set_oc(xref, ocxref)`

- New in v1.18.4

If *xref* represents an image or form xobject, set or remove the cross reference number *ocxref* of an optional contents object.

Parameters

- **xref** (*int*) – the *xref* of an image or form xobject⁵. Valid such cross reference numbers are returned by `Document.get_page_images()`, resp. `Document.get_page_xobjects()`. For invalid numbers, an exception is raised.
- **ocxref** (*int*) – the *xref* number of an *OCG* / *OCMD*. If not zero, an invalid reference raises an exception. If zero, any OC reference is removed.

`get_layers()`

- New in v1.18.3

⁵ Examples for “Form XObjects” are created by `Page.show_pdf_page()`.

Show optional layer configurations. There always is a standard one, which is not included in the response.

```
>>> for item in doc.get_layers(): print(item)
{'number': 0, 'name': 'my-config', 'creator': ''}
>>> # use 'number' as config identifier in add_ocg
```

add_layer(*name*, *creator*=None, *on*=None)

- New in v1.18.3

Add an optional content configuration. Layers serve as a collection of ON / OFF states for optional content groups and allow fast visibility switches between different views on the same document.

Parameters

- **name** (*str*) – arbitrary name.
- **creator** (*str*) – (optional) creating software.
- **on** (*sequ*) – a sequence of OCG [xref](#) numbers which should be set to ON when this layer gets activated. All OCGs not listed here will be set to OFF.

switch_layer(*number*, *as_default*=False)

- New in v1.18.3

Switch to a document view as defined by the optional layer's configuration number. This is temporary, except if established as default.

Parameters

- **number** (*int*) – config number as returned by `Document.layer_configs()`.
- **as_default** (*bool*) – make this the default configuration.

Activates the ON / OFF states of OCGs as defined in the identified layer. If *as_default*=True, then additionally all layers, including the standard one, are merged and the result is written back to the standard layer, and **all optional layers are deleted**.

add_ocg(*name*, *config*=-1, *on*=True, *intent*='View', *usage*='Artwork')

- New in v1.18.3

Add an optional content group. An OCG is the most important unit of information to determine object visibility. For a PDF, in order to be regarded as having optional content, at least one OCG must exist.

Parameters

- **name** (*str*) – arbitrary name. Will show up in supporting PDF viewers.
- **config** (*int*) – layer configuration number. Default -1 is the standard configuration.
- **on** (*bool*) – standard visibility status for objects pointing to this OCG.
- **intent** (*str*, *list*) – a string or list of strings declaring the visibility intents. There are two PDF standard values to choose from: “View” and “Design”. Default is “View”. **Correct spelling is important**.
- **usage** (*str*) – another influencer for OCG visibility. This will become part of the OCG's /Usage key. There are two PDF standard values to choose from: “Artwork” and “Technical”. Default is “Artwork”. Please only change when required.

Returns

[xref](#) of the created OCG. Use as entry for oc parameter in supporting objects.

Note: Multiple OCGs with identical parameters may be created. This will not cause problems. Garbage option 3 of `Document.save()` will get rid of any duplicates.

set_ocmd(*xref*=0, *ocgs*=None, *policy*='AnyOn', *ve*=None)

- New in v1.18.4

Create or update an [OCMD](#), **Optional Content Membership Dictionary**.

Parameters

- **xref** (*int*) – [xref](#) of the OCMD to be updated, or 0 for a new OCMD.
- **ocgs** (*list*) – a sequence of [xref](#) numbers of existing [OCG](#) PDF objects.
- **policy** (*str*) – one of “AnyOn” (default), “AnyOff”, “AllOn”, “AllOff” (mixed or lower case).
- **ve** (*list*) – a “visibility expression”. This is a list of arbitrarily nested other lists – see explanation below. Use as an alternative to the combination *ocgs* / *policy* if you need to formulate more complex conditions.

Return type

int

Returns

[xref](#) of the OCMD. Use as `oc=xref` parameter in supporting objects, and respectively in `Document.set_oc()` or `Annot.set_oc()`.

Note: Like an OCG, an OCMD has a visibility state ON or OFF, and it can be used like an OCG. In contrast to an OCG, the OCMD state is determined by evaluating the state of one or more OCGs via special forms of **boolean expressions**. If the expression evaluates to true, the OCMD state is ON and OFF for false.

There are two ways to formulate OCMD visibility:

1. Use the combination of *ocgs* and *policy*: The *policy* value is interpreted as follows:
 - AnyOn – (default) true if at least one OCG is ON.
 - AnyOff – true if at least one OCG is OFF.
 - AllOn – true if all OCGs are ON.
 - AllOff – true if all OCGs are OFF.

Suppose you want two PDF objects be displayed exactly one at a time (if one is ON, then the other one must be OFF):

Solution: use an **OCG** for object 1 and an **OCMD** for object 2. Create the OCMD via `set_ocmd(ocgs=[xref], policy="AllOff")`, with the [xref](#) of the OCG.

2. Use the **visibility expression** *ve*: This is a list of two or more items. The **first item** is a logical keyword: one of the strings “and”, “or”, or “not”. The **second** and all subsequent items must either be an integer or another list. An integer must be the [xref](#) number of an OCG. A list must again have at least two items starting with one of the boolean keywords. This syntax is a bit awkward, but quite powerful:
 - Each list must start with a logical keyword.
 - If the keyword is a “not”, then the list must have exactly two items. If it is “and” or “or”, any number of other items may follow.
 - Items following the logical keyword may be either integers or again a list. An *integer* must be the xref of an OCG. A *list* must conform to the previous rules.

Examples:

- `set_ocmd(ve=["or", 4, ["not", 5], ["and", 6, 7]])`. This delivers ON if the following is true: “4 is ON, or 5 is OFF, or 6 and 7 are both ON”.
- `set_ocmd(ve=["not", xref])`. This has the same effect as the OCMD example created under 1.

For more details and examples see page 224 of [Adobe PDF References](#). Also do have a look at example scripts [here](#).

Visibility expressions, /VE, are part of PDF specification version 1.6. So not all PDF viewers / readers may already support this feature and hence will react in some standard way for those cases.

get_ocmd(*xref*)

- New in v1.18.4

Retrieve the definition of an *OCMD*.

Parameters

xref (*int*) – the *xref* of the OCMD.

Return type

dict

Returns

a dictionary with the keys *xref*, *ocgs*, *policy* and *ve*.

get_layer(*config=-1*)

- New in v1.18.3

List of optional content groups by status in the specified configuration. This is a dictionary with lists of cross reference numbers for OCGs that occur in the arrays */ON*, */OFF* or in some radio button group (*/RBGroups*).

Parameters

config (*int*) – the configuration layer (default is the standard config layer).

```
>>> pprint(doc.get_layer())
{'off': [8, 9, 10], 'on': [5, 6, 7], 'rbgroups': [[7, 10]]}
>>>
```

set_layer(*config*, *on=None*, *off=None*, *basestate=None*, *rbgroups=None*)

- New in v1.18.3

Mass status changes of optional content groups. **Permanently** sets the status of OCGs.

Parameters

- **config** (*int*) – desired configuration layer, choose -1 for the default one.
- **on** (*list*) – list of *xref* of OCGs to set ON. Replaces previous values. An empty list will cause no OCG being set to ON anymore. Should be specified if *basestate="ON"* is used.
- **off** (*list*) – list of *xref* of OCGs to set OFF. Replaces previous values. An empty list will cause no OCG being set to OFF anymore. Should be specified if *basestate="OFF"* is used.
- **basestate** (*str*) – desired state of OCGs that are not mentioned in *on* resp. *off*. Possible values are “ON”, “OFF” or “Unchanged”. Upper / lower case possible.
- **rbgroups** (*list*) – a list of lists. Replaces previous values. Each sublist should contain two or more OCG xrefs. OCGs in the same sublist are handled like buttons in a radio button group: setting one to ON automatically sets all other group members to OFF.

Values *None* will not change the corresponding PDF array.

```
>>> doc.set_layer(-1, basestate="OFF") # only changes the base state
>>> pprint(doc.get_layer())
{'basestate': 'OFF', 'off': [8, 9, 10], 'on': [5, 6, 7], 'rbgroups':
↳ [[7, 10]]}
```

get_ocgs()

- New in v1.18.3

Details of all optional content groups. This is a dictionary of dictionaries like this (key is the OCG's *xref*):

```
>>> pprint(doc.get_ocgs())
{13: {'on': True,
```

(continues on next page)

(continued from previous page)

```

        'intent': ['View', 'Design'],
        'name': 'Circle',
        'usage': 'Artwork'},
14: {'on': True,
    'intent': ['View', 'Design'],
    'name': 'Square',
    'usage': 'Artwork'},
15: {'on': False, 'intent': ['View'], 'name': 'Square', 'usage':
    ↪ 'Artwork'}}
>>>

```

layer_ui_configs()

- New in v1.18.3

Show the visibility status of optional content that is modifyable by the user interface of supporting PDF viewers. Example:

```

>>> pprint(doc.layer_ui_configs())
({'depth': 0,
  'locked': False,
  'number': 0,
  'on': True,
  'text': 'Circle',
  'type': 'checkbox'},
 {'depth': 0,
  'locked': False,
  'number': 1,
  'on': False,
  'text': 'Square',
  'type': 'checkbox'})
>>> # refers to OCGs named "Circle" (ON), resp. "Square" (OFF)

```

Note:

- Only reports items contained in the currently selected layer configuration.
- **The meaning of the dictionary keys is as follows:**
 - *depth*: item's nesting level in the */Order* array
 - *locked*: whether changing the item's state is prohibited
 - *number*: running sequence number
 - *on*: item state
 - *text*: text string or name field of the originating OCG
 - *type*: one of “label” (set by a text string), “checkbox” (set by a single OCG) or “radiobox” (set by a set of connected OCGs)

set_layer_ui_config(number, action=0)

- New in v1.18.3

Modify OC visibility status of content groups. This is analog to what supporting PDF viewers would offer.

Note: Visibility is **not** a property stored with the OCG. It is not even an information necessarily present in the PDF document at all. Instead, the current visibility is **temporarily** set using the

user interface of some supporting PDF consumer software. The same type of functionality is offered by this method.

To make **permanent** changes, use `Document.set_layer()`.

Parameters

- **number** (*in*) – number as returned by `Document.layer_ui_configs()`.
- **action** (*int*) – 0 = set on (default), 1 = toggle on/off, 2 = set off.

Example:

```
>>> # let's make above "Square" visible:
>>> doc.set_layer_ui_config(1, action=0)
>>> pprint(doc.layer_ui_configs())
({'depth': 0,
  'locked': False,
  'number': 0,
  'on': True,
  'text': 'Circle',
  'type': 'checkbox'},
 {'depth': 0,
  'locked': False,
  'number': 1,
  'on': True, # <===
  'text': 'Square',
  'type': 'checkbox'})
>>>
```

authenticate(*password*)

Decrypts the document with the string *password*. If successful, document data can be accessed. For PDF documents, the “owner” and the “user” have different privileges, and hence different passwords may exist for these authorization levels. The method will automatically establish the appropriate (owner or user) access rights for the provided password.

Parameters

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

Return type

`int`

Returns

a positive value if successful, zero otherwise (the string does not match either password). If positive, the indicator `Document.is_encrypted` is set to `False`.

Positive return codes carry the following information detail:

- 1 => authenticated, but the PDF has neither owner nor user passwords.
- 2 => authenticated with the **user** password.
- 4 => authenticated with the **owner** password.
- 6 => authenticated and both passwords are equal – probably a rare situation.

Note: The document may be protected by an owner, but **not** by a user password. Detect this situation via `doc.authenticate("") == 2`. This allows opening and

reading the document without authentication, but, depending on the `Document.permissions` value, other actions may be prohibited. PyMuPDF (like MuPDF) in this case **ignores those restrictions**. So, – in contrast to any PDF viewers – you can for example extract text and add or modify content, even if the respective permission flags `PDF_PERM_COPY`, `PDF_PERM_MODIFY`, `PDF_PERM_ANNOTATE`, etc. are set off! It is your responsibility building a legally compliant application where applicable.

get_page_numbers(*label*, *only_one=False*)

- New in v 1.18.6

PDF only: Return a list of page numbers that have the specified label – note that labels may not be unique in a PDF. This implies a sequential search through **all page numbers** to compare their labels.

Note: Implementation detail – pages are **not loaded** for this purpose.

Parameters

- **label** (*str*) – the label to look for, e.g. “vii” (Roman number 7).
- **only_one** (*bool*) – stop after first hit. Useful e.g. if labelling is known to be unique, or there are many pages, etc. The default will check every page number.

Return type

list

Returns

list of page numbers that have this label. Empty if none found, no labels defined, etc.

get_page_labels()

- New in v1.18.7

PDF only: Extract the list of page label definitions. Typically used for modifications before feeding it into `Document.set_page_labels()`.

Returns

a list of dictionaries as defined in `Document.set_page_labels()`.

set_page_labels(*labels*)

- New in v1.18.6

PDF only: Add or update the page label definitions of the PDF.

Parameters

labels (*list*) – a list of dictionaries. Each dictionary defines a label building rule and a 0-based “start” page number. That start page is the first for which the label definition is valid. Each dictionary has up to 4 items and looks like `{'startpage': int, 'prefix': str, 'style': str, 'firstpagenum': int}` and has the following items.

- **startpage**: (int) the first page number (0-based) to apply the label rule. This key **must be present**. The rule is applied to all subsequent pages until either end of document or superseded by the rule with the next larger page number.

- **prefix:** (str) an arbitrary string to start the label with, e.g. “A-”. Default is “”.
- **style:** (str) the numbering style. Available are “D” (decimal), “r”/“R” (Roman numbers, lower / upper case), and “a”/“A” (lower / upper case alphabetical numbering: “a” through “z”, then “aa” through “zz”, etc.). Default is “”. If “”, no numbering will take place and the pages in that range will receive the same label consisting of the **prefix** value. If **prefix** is also omitted, then the label will be “”.
- **firstpagenum:** (int) start numbering with this value. Default is 1, smaller values are ignored.

For example:

```
[{'startpage': 6, 'prefix': 'A-', 'style': 'D', 'firstpagenum': 10},  
 {'startpage': 10, 'prefix': '', 'style': 'D', 'firstpagenum': 1}]
```

will generate the labels “A-10”, “A-11”, “A-12”, “A-13”, “1”, “2”, “3”, ... for pages 6, 7 and so on until end of document. Pages 0 through 5 will have the label “”.

make_bookmark(*loc*)

- New in v.1.17.3

Return a page pointer in a reflowable document. After re-layouting the document, the result of this method can be used to find the new location of the page.

Note: Do not confuse with items of a table of contents, TOC.

Parameters

loc (*list*, *tuple*) – page location. Must be a valid (*chapter*, *pno*).

Return type

pointer

Returns

a long integer in pointer format. To be used for finding the new location of the page after re-layouting the document. Do not touch or re-assign.

find_bookmark(*bookmark*)

- New in v.1.17.3

Return the new page location after re-layouting the document.

Parameters

bookmark (*pointer*) – created by `Document.make_bookmark()`.

Return type

tuple

Returns

the new (*chapter*, *pno*) of the page.

chapter_page_count(*chapter*)

- New in v.1.17.0

Return the number of pages of a chapter.

Parameters

chapter (*int*) – the 0-based chapter number.

Return type

int

Returns

number of pages in chapter. Relevant only for document types which chapter support (EPUB currently).

next_location(*page_id*)

- New in v.1.17.0

Return the location of the following page.

Parameters

page_id (*tuple*) – the current page id. This must be a tuple (*chapter*, *pno*) identifying an existing page.

Returns

The tuple of the following page, i.e. either (*chapter*, *pno* + 1) or (*chapter* + 1, 0), **or** the empty tuple () if the argument was the last page. Relevant only for document types which chapter support (EPUB currently).

prev_location(*page_id*)

- New in v.1.17.0

Return the locator of the preceeding page.

Parameters

page_id (*tuple*) – the current page id. This must be a tuple (*chapter*, *pno*) identifying an existing page.

Returns

The tuple of the preceeding page, i.e. either (*chapter*, *pno* - 1) or the last page of the receeding chapter, **or** the empty tuple () if the argument was the first page. Relevant only for document types which chapter support (EPUB currently).

load_page(*page_id*=0)

- Changed in v1.17.0: For document types supporting a so-called “chapter structure” (like EPUB), pages can also be loaded via the combination of chapter number and relative page number, instead of the absolute page number. This should **significantly speed up access** for large documents.

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

Parameters

page_id (*int*, *tuple*) – (*Changed in v1.17.0*)

Either a 0-based page number, or a tuple (*chapter*, *pno*). For an **integer**, any $-\infty < \text{page_id} < \text{page_count}$ is acceptable. While *page_id* is negative, [page_count](#) will be added to it. For example: to load the last page, you can use `doc.load_page(-1)`. After this you have `page.number = doc.page_count - 1`.

For a tuple, *chapter* must be in range `Document.chapter_count`, and *pno* must be in range `Document.chapter_page_count()` of that chapter. Both values are 0-based. Using this notation, [Page.number](#) will equal the given tuple. Relevant only for document types which chapter support (EPUB currently).

Return type*Page*

Note: Documents also follow the Python sequence protocol with page numbers as indices: `doc.load_page(n) == doc[n]`.

For **absolute page numbers** only, expressions like “*for page in doc: ...*” and “*for page in reversed(doc): ...*” will successively yield the document’s pages. Refer to `Document.pages()` which allows processing pages as with slicing.

You can also use index notation with the new chapter-based page identification: use `page = doc[(5, 2)]` to load the third page of the sixth chapter.

To maintain a consistent API, for document types not supporting a chapter structure (like PDFs), `Document.chapter_count` is 1, and pages can also be loaded via tuples `(0, pno)`. See this³ footnote for comments on performance improvements.

reload_page(*page*)

- New in v1.16.10

PDF only: Provide a new copy of a page after finishing and updating all pending changes.

Parameters

page (*Page*) – page object.

Return type*Page***Returns**

a new copy of the same page. All pending updates (e.g. to annotations or widgets) will be finalized and a fresh copy of the page will be loaded.

Note: In a typical use case, a page *Pixmap* should be taken after annotations / widgets have been added or changed. To force all those changes being reflected in the page structure, this method re-instates a fresh copy while keeping the object hierarchy “document -> page -> annotations/widgets” intact.

page_croptbox(*pno*)

- New in v1.17.7

PDF only: Return the unrotated page rectangle – **without loading the page** (via `Document.load_page()`). This is meant for internal purpose requiring best possible performance.

Parameters

pno (*int*) – 0-based page number.

Returns

Rect of the page like `Page.rect()`, but ignoring any rotation.

page_xref(*pno*)

- New in v1.17.7

³ For applicable (EPUB) document types, loading a page via its absolute number may result in layouting a large part of the document, before the page can be accessed. To avoid this performance impact, prefer chapter-based access. Use convenience methods and attributes `Document.next_location()`, `Document.prev_location()` and `Document.last_location` for maintaining a high level of coding efficiency.

PDF only: Return the [xref](#) of the page – **without loading the page** (via [Document.load_page\(\)](#)). This is meant for internal purpose requiring best possible performance.

Parameters

pno (*int*) – 0-based page number.

Returns

[xref](#) of the page like [Page.xref](#).

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

- New in v1.16.4

A generator for a range of pages. Parameters have the same meaning as in the built-in function [range\(\)](#). Intended for expressions of the form “*for page in doc.pages(start, stop, step): ...*”.

Parameters

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

Returns

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

- “`doc.pages()`” emits all pages.
- “`doc.pages(4, 9, 2)`” emits pages 4, 6, 8.
- “`doc.pages(0, None, 2)`” emits all pages with even numbers.
- “`doc.pages(-2)`” emits the last two pages.
- “`doc.pages(-1, -1)`” emits all pages in reversed order.
- “`doc.pages(-1, -10)`” always emits 10 pages in reversed order, starting with the last page – **repeatedly** if the document has less than 10 pages. So for a 4-page document the following page numbers are emitted: 3, 2, 1, 0, 3, 2, 1, 0, 3, 2, 1, 0, 3.

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

Create a PDF version of the current document and write it to memory. **All document types** are supported. The parameters have the same meaning as in [insert_pdf\(\)](#). 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 `tobytes(garbage=4, deflate=True)`. See `tobytes()`. You can output it directly to disk or open it as a PDF. Here are some examples:

```
>>> # convert an XPS file to PDF
>>> xps = fitz.open("some.xps")
>>> pdfbytes = xps.convert_to_pdf()
>>>
>>> # either do this -->
>>> pdf = fitz.open("pdf", pdfbytes)
>>> pdf.save("some.pdf")
>>>
>>> # or this -->
>>> pdfout = open("some.pdf", "wb")
>>> pdfout.tobytes(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.convert_to_pdf() # make a 1-page_
↳ PDF of it
>>>     imgpdf=fitz.open("pdf", pdfbytes)
>>>     doc.insert_pdf(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.insert_image()` instead. Otherwise, this method is recommended for its much better performance.
 - XPS: appearance very good. Links work fine, outlines (bookmarks) are lost, but can easily be recovered².
 - EPUB, CBZ, FB2: similar to XPS.
 - SVG: medium. Roughly comparable to `svglib`.
-

`get_toc(simple=True)`

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

Parameters

simple (*bool*) – Indicates whether a simple or a detailed TOC is required. If

² However, you **can** use `Document.get_toc()` and `Page.get_links()` (which are available for all document types) and copy this information over to the output PDF. See demo `pdf-converter.py`.

False, each item 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*). If *-1* either no destination or outside document.
- *dest* – (*dict*) included only if *simple=False*. Contains details of the TOC item as follows:
 - *kind*: destination kind, see *Link Destination Kinds*.
 - *file*: filename if kind is *LINK_GOTOR* or *LINK_LAUNCH*.
 - *page*: target page, 0-based, *LINK_GOTOR* or *LINK_GOTO* only.
 - *to*: position on target page (*Point*).
 - *zoom*: (float) zoom factor on target page.
 - *xref*: *xref* of the item (0 if no PDF).
 - *color*: item color in PDF RGB format (red, green, blue), or omitted (always omitted if no PDF).
 - *bold*: true if bold item text or omitted. PDF only.
 - *italic*: true if italic item text, or omitted. PDF only.
 - *collapse*: true if sub-items are folded, or omitted. PDF only.

xref_get_keys(*xref*)

- New in v1.18.7

PDF only: Return the PDF dictionary keys of the *dictionary* object provided by its xref number.

Parameters

xref (*int*) – the *xref*. (Changed in v1.18.10) Use *-1* to access the special dictionary “PDF trailer”.

Returns

a tuple of dictionary keys present in object *xref*. Examples:

```
>>> from pprint import pprint
>>> import fitz
>>> doc=fitz.open("pymupdf.pdf")
>>> xref = doc.page_xref(0) # xref of page 0
>>> pprint(doc.xref_get_keys(xref)) # primary level keys of
↳ a page
('Type', 'Contents', 'Resources', 'MediaBox', 'Parent')
```

(continues on next page)

(continued from previous page)

```
>>> pprint(doc.xref_get_keys(-1)) # primary level keys of
↳ the trailer
('Type', 'Index', 'Size', 'W', 'Root', 'Info', 'ID', 'Length
↳ ', 'Filter')
>>>
```

xref_get_key(xref, key)

- New in v1.18.7

PDF only: Return type and value of a PDF dictionary key of a *dictionary* object given by its xref.

Parameters

- **xref** (*int*) – the *xref*. *Changed in v1.18.10:* Use -1 to access the special dictionary “PDF trailer”.
- **key** (*str*) – the desired PDF key. Must **exactly** match (case-sensitive) one of the keys contained in *Document.xref_get_keys()*.

Return type

tuple

Returns

A tuple (type, value) of strings, where type is one of “xref”, “array”, “dict”, “int”, “float”, “null”, “bool”, “name”, “string” or “unknown” (should not occur). Independent of “type”, the value of the key is **always** formatted as a string – see the following example – and (almost always) a faithful reflection of what is stored in the PDF. In most cases, the format of the value string also gives a clue about the key type:

- A “name” always starts with a “/” slash.
- An “xref” always ends with “ 0 R”.
- An “array” is always enclosed in “[...]” brackets.
- A “dict” is always enclosed in “<<...>>” brackets.
- A “bool”, resp. “null” always equal either “true”, “false”, resp. “null”.
- “float” and “int” are represented by their string format – and are thus not always distinguishable.
- A “string” is converted to UTF-8 and may therefore deviate from what is stored in the PDF. For example, the PDF key “Author” may have a value of “<FEFF004A006F0072006A00200058002E0020004D0063004B00690065>” in the file, but the method will return ('string', 'Jorj X. McKie').

```
>>> for key in doc.xref_get_keys(xref):
        print(key, "=", doc.xref_get_key(xref, key))
Type = ('name', '/Page')
Contents = ('xref', '1297 0 R')
Resources = ('xref', '1296 0 R')
MediaBox = ('array', '[0 0 612 792]')
Parent = ('xref', '1301 0 R')
>>> #
```

(continues on next page)

(continued from previous page)

```

>>> # Now same thing for the PDF trailer.
>>> # It has no xref, so -1 must be used instead.
>>> #
>>> for key in doc.xref_get_keys(-1):
>>>     print(key, "=", doc.xref_get_key(-1, key))
Type = ('name', '/XRef')
Index = ('array', '[0 8802]')
Size = ('int', '8802')
W = ('array', '[1 3 1]')
Root = ('xref', '8799 0 R')
Info = ('xref', '8800 0 R')
ID = ('array', ' [<DC9D56A6277EFFF82084E64F9441E18C>
-><DC9D56A6277EFFF82084E64F9441E18C> ] ')
Length = ('int', '21111')
Filter = ('name', '/FlateDecode')
>>>

```

xref_set_key(xref, key, value)

- New in v1.18.7, changed in v 1.18.13
- Changed in v1.19.4: remove a key “physically” if set to “null”.

PDF only: Set (add, update, delete) the value of a PDF key for the *dictionary* object given by its xref.

Caution: This is an expert function: if you do not know what you are doing, there is a high risk to render (parts of) the PDF unusable. Please do consult *Adobe PDF References* about object specification formats (page 18) and the structure of special dictionary types like page objects.

Parameters

- **xref** (*int*) – the *xref*. *Changed in v1.18.13:* To update the PDF trailer, specify -1.
- **key** (*str*) – the desired PDF key (without leading “/”). Must not be empty. Any valid PDF key – whether already present in the object (which will be overwritten) – or new. It is possible to use PDF path notation like “Resources/ExtGState” – which sets the value for key “/ExtGState” as a sub-object of “/Resources”.
- **value** (*str*) – the value for the key. It must be a non-empty string and, depending on the desired PDF object type, the following rules must be observed. There is some syntax checking, but **no type checking** and no checking if it makes sense PDF-wise, i.e. **no semantics checking**. Upper / lower case is important!
- **xref** – must be provided as “nnn 0 R” with a valid *xref* number nnn of the PDF. The suffix “0 R” is required to be recognizable as an xref by PDF applications.
- **array** – a string like “[a b c d e f]”. The brackets are required. Array items must be separated by at least one space (not commas like in Python). An empty array “[]” is possible and *equivalent* to removing the key. Array items may be any PDF objects, like dictionaries, xrefs, other arrays, etc. Like in Python, array items may be of different types.

- **dict** – a string like "<< . . . >>". The brackets are required and must enclose a valid PDF dictionary definition. The empty dictionary "<<>>" is possible and *equivalent* to removing the key.
- **int** – an integer formatted **as a string**.
- **float** – a float formatted **as a string**. Scientific notation (with exponents) is **not allowed by PDF**.
- **null** – the string "null". This is the PDF equivalent to Python's None and causes the key to be ignored – however not necessarily removed, resp. removed on saves with garbage collection. *Changed in v1.19.4:* If the key is no path hierarchy (i.e. contains no slash “/”), then it will be completely removed.
- **bool** – one of the strings "true" or "false".
- **name** – a valid PDF name with a leading slash: "/PageLayout". See page 16 of the [Adobe PDF References](#).
- **string** – a valid PDF string. **All PDF strings must be enclosed by brackets**. Denote the empty string as "()". Depending on its content, the possible brackets are
 - “(…)” for ASCII-only text. Reserved PDF characters must be backslash-escaped and non-ASCII characters must be provided as 3-digit backslash-escaped octals – including leading zeros. Example: 12 = 0x0C must be encoded as \014.
 - “<...>” for hex-encoded text. Every character must be represented by two hex-digits (lower or upper case).
 - If in doubt, we **strongly recommend** to use [get_pdf_str\(\)](#)! This function automatically generates the right brackets, escapes, and overall format. It will for example do conversions like these:

```
>>> # because of the € symbol, the following yields UTF-16BE BOM
>>> fitz.get_pdf_str("Pay in $ or €.")
'<feff00500061007900200069006e002000240020006f0072002020ac002e>'
>>> # escapes for brackets and non-ASCII
>>> fitz.get_pdf_str("Prices in EUR (USD also accepted). Areas_
↪are in m².")
'(Prices in EUR \\\(USD also accepted\\). Areas are in m\\262.)'
```

get_page_pixmap(*pno*: int, *, *matrix*: *matrix_like* = Identity, *dpi*=None, *colorspace*: Colorspace = csRGB, *clip*: *rect_like* = None, *alpha*: bool = False, *annots*: bool = True)

Creates a pixmap from page *pno* (zero-based). Invokes [Page.get_pixmap\(\)](#).

All parameters except *pno* are *keyword-only*.

Parameters

pno (*int*) – page number, 0-based in $-\infty < \text{pno} < \text{page_count}$.

Return type

[Pixmap](#)

get_page_xobjects(*pno*)

- New in v1.16.13
- Changed in v1.18.11

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

Parameters

pno (*int*) – page number, 0-based, $-\infty < \text{pno} < \text{page_count}$.

Return type

list

Returns

a list of (non-image) XObjects. These objects typically represent pages *embedded* (not copied) from other PDFs. For example, `Page.show_pdf_page()` will create this type of object. An item of this list has the following layout: (**xref**, **name**, **invoker**, **bbox**), where

- **xref** (*int*) is the XObject's *xref*.
- **name** (*str*) is the symbolic name to reference the XObject.
- **invoker** (*int*) the *xref* of the invoking XObject or zero if the page directly invokes it.
- **bbox** (*Rect*) the boundary box of the XObject's location on the page **in untransformed coordinates**. To get actual, non-rotated page coordinates, multiply with the page's transformation matrix `Page.transformation_matrix`. *Changed in v.18.11: the bbox is now formatted as *Rect*.*

`get_page_images(pno, full=False)`

PDF only: Return a list of all images (directly or indirectly) referenced by the page.

Parameters

- **pno** (*int*) – page number, 0-based, $-\infty < \text{pno} < \text{page_count}$.
- **full** (*bool*) – whether to also include the referencer's *xref* (which is zero if this is the page).

Return type

list

Returns

a list of images **referenced** by this page. Each item looks like
(**xref**, **smask**, **width**, **height**, **bpc**, **colorspace**, **alt**,
colorspace, **name**, **filter**, **referencer**)

Where

- **xref** (*int*) is the image object number
- **smask** (*int*) is the object number of its soft-mask image
- **width** and **height** (*ints*) are the image dimensions
- **bpc** (*int*) denotes the number of bits per component (normally 8)
- **colorspace** (*str*) a string naming the colorspace (like **DeviceRGB**)
- **alt. colorspace** (*str*) is any alternate colorspace depending on the value of **colorspace**
- **name** (*str*) is the symbolic name by which the image is referenced
- **filter** (*str*) is the decode filter of the image (*Adobe PDF References*, pp. 22).
- **referencer** (*int*) the *xref* of the referencer. Zero if directly referenced by the page. Only present if *full=True*.

Note: In general, this is not the list of images that are **actually displayed**. This method only parses several PDF objects to collect references to embedded images. It does not analyse the page’s *contents*, where all the actual image display commands are defined. To get this information, please use `Page.get_image_info()`. Also have a look at the discussion in section *Structure of Dictionary Outputs*.

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

PDF only: Return a list of all fonts (directly or indirectly) referenced by the page.

Parameters

- **pno** (*int*) – page number, 0-based, $-\infty < \text{pno} < \text{page_count}$.
- **full** (*bool*) – whether to also include the referencer’s *xref*. If *True*, the returned items are one entry longer. Use this option if you need to know, whether the page directly references the font. In this case the last entry is 0. If the font is referenced by an *XObject* of the page, you will find its *xref* here.

Return type

list

Returns

a list of fonts referenced by this page. Each entry looks like

(*xref*, *ext*, *type*, *basefont*, *name*, *encoding*, *referencer*),

where

- **xref** (*int*) is the font object number (may be zero if the PDF uses one of the builtin fonts directly)
- **ext** (*str*) font file extension (e.g. “ttf”, see *Font File Extensions*)
- **type** (*str*) is the font type (like “Type1” or “TrueType” etc.)
- **basefont** (*str*) is the base font name,
- **name** (*str*) is the symbolic name, by which the font is referenced
- **encoding** (*str*) the font’s character encoding if different from its built-in encoding (*Adobe PDF References*, p. 254):
- **referencer** (*int* optional) the *xref* of the referencer. Zero if directly referenced by the page, otherwise the xref of an *XObject*. Only present if *full=True*.

Example:

```
>>> pprint(doc.get_page_fonts(0, full=False))
[(12, 'ttf', 'TrueType', 'FNUUTH+Calibri-Bold', 'R8', ''),
 (13, 'ttf', 'TrueType', 'DOKBTG+Calibri', 'R10', ''),
 (14, 'ttf', 'TrueType', 'NOHSJV+Calibri-Light', 'R12', ''),
 (15, 'ttf', 'TrueType', 'NZNDCL+CourierNewPSMT', 'R14', ''),
 (16, 'ttf', 'Type0', 'MNCSJY+SymbolMT', 'R17', 'Identity-H'),
 (17, 'cff', 'Type1', 'UAEUYH+Helvetica', 'R20', 'WinAnsiEncoding'),
 (18, 'ttf', 'Type0', 'ECPLRU+Calibri', 'R23', 'Identity-H'),
 (19, 'ttf', 'Type0', 'TONAYT+CourierNewPSMT', 'R27', 'Identity-H')]
```

Note:

- This list has no duplicate entries: the combination of *xref*, *name* and *referencer* is unique.
 - In general, this is a superset of the fonts actually in use by this page. The PDF creator may e.g. have specified some global list, of which each page only makes partial use.
-

get_page_text(*pno*, *output*='text', *flags*=3, *textpage*=None, *sort*=False)

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

Parameters

pno (*int*) – page number, 0-based, any value $-\infty < \text{pno} < \text{page_count}$.

For other parameter refer to the page method.

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 *is_reflowable*.

Parameters

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

select(*s*)

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

Parameters

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

Note:

- Page numbers in the sequence need not be unique nor be in any particular order. This makes the method a versatile utility to e.g. select only the even or the odd pages or meeting some other criteria and so forth.
 - On a technical level, the method will always create a new *pagetree*.
 - When dealing with only a few pages, methods *copy_page()*, *move_page()*, *delete_page()* are easier to use. In fact, they are also **much faster** – by at least one order of magnitude when the document has many pages.
-

set_metadata(*m*)

PDF only: Sets or updates the metadata of the document as specified in *m*, a Python dictionary.

Parameters

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

(*Changed in v1.18.4*) Empty values or “none” are no longer written, but completely omitted.

get_xml_metadata()

PDF only: Get the document XML metadata.

Return type

str

Returns

XML metadata of the document. Empty string if not present or not a PDF.

set_xml_metadata(xml)

PDF only: Sets or updates XML metadata of the document.

Parameters

xml (*str*) – the new XML metadata. Should be XML syntax, however no checking is done by this method and any string is accepted.

set_toc(toc, collapse=1)

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

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

Parameters

- **toc** (*sequence*) – A list / tuple with **all bookmark entries** that should form the new table of contents. Output variants of [get_toc\(\)](#) are acceptable. To completely remove the table of contents specify an empty sequence or *None*. Each item must be a list with the following format.

– [lvl, title, page [, dest]] where

- * **lvl** is the hierarchy level (int > 0) of the item, which **must be 1** for the first item and at most 1 larger than the previous one.
- * **title** (str) is the title to be displayed. It is assumed to be UTF-8-encoded (relevant for multibyte code points only).
- * **page** (int) is the target page number (**attention: 1-based**). Must be in valid range if positive. Set it to -1 if there is no target, or the target is external.
- * **dest** (optional) is a dictionary or a number. If a number, it will be interpreted as the desired height (in points) this entry should point to on the page. Use a dictionary (like the one given as output by [get_toc\(False\)](#)) for a detailed control of the bookmark’s properties, see [Document.get_toc\(\)](#) for a description.

- **collapse** (*int*) – (new in v1.16.9) controls the hierarchy level beyond which outline entries should initially show up collapsed. The default 1 will hence only display level 1, higher levels must be unfolded using the PDF viewer. To unfold everything, specify either a large integer, 0 or None.

Return type

int

Returns

the number of inserted, resp. deleted items.

outline_xref(*idx*)

- New in v1.17.7

PDF only: Return the [xref](#) of the outline item. This is mainly used for internal purposes.

arg int *idx*: index of the item in list [Document.get_toc\(\)](#).

Returns[xref](#).**del_toc_item(*idx*)**

- New in v1.17.7
- Changed in v1.18.14: no longer remove the item's text, but show it grayed-out.

PDF only: Remove this TOC item. This is a high-speed method, which **disables** the respective item, but leaves the overall TOC struture intact. Physically, the item still exists in the TOC tree, but is shown grayed-out and will no longer point to any destination.

This also implies that you can reassign the item to a new destination using [Document.set_toc_item\(\)](#), when required.

Parameters

idx (*int*) – the index of the item in list [Document.get_toc\(\)](#).

set_toc_item(*idx*, *dest_dict*=None, *kind*=None, *pno*=None, *uri*=None, *title*=None, *to*=None, *filename*=None, *zoom*=0)

- New in v1.17.7
- Changed in v1.18.6

PDF only: Changes the TOC item identified by its index. Change the item **title**, **destination**, **appearance** (color, bold, italic) or collapsing sub-items – or to remove the item altogether.

Use this method if you need specific changes for selected entries only and want to avoid replacing the complete TOC. This is beneficial especially when dealing with large table of contents.

Parameters

- **idx** (*int*) – the index of the entry in the list created by [Document.get_toc\(\)](#).
- **dest_dict** (*dict*) – the new destination. A dictionary like the last entry of an item in [doc.get_toc\(False\)](#). Using this as a template is recommended. When given, **all other parameters are ignored** – except title.
- **kind** (*int*) – the link kind, see [Link Destination Kinds](#). If [LINK_NONE](#), then all remaining parameter will be ignored, and the TOC item will be removed – same as [Document.del_toc_item\(\)](#). If None, then only the

title is modified and the remaining parameters are ignored. All other values will lead to making a new destination dictionary using the subsequent arguments.

- **pno** (*int*) – the 1-based page number, i.e. a value $1 \leq \text{pno} \leq \text{doc.page_count}$. Required for LINK_GOTO.
- **uri** (*str*) – the URL text. Required for LINK_URI.
- **title** (*str*) – the desired new title. None if no change.
- **to** (*point_like*) – (optional) points to a coordinate on the target page. Relevant for LINK_GOTO. If omitted, a point near the page's top is chosen.
- **filename** (*str*) – required for LINK_GOTOR and LINK_LAUNCH.
- **zoom** (*float*) – use this zoom factor when showing the target page.

Example use: Change the TOC of the SWIG manual to achieve this:

Collapse everything below top level and show the chapter on Python support in red, bold and italic:

```
>>> import fitz
>>> doc=fitz.open("SWIGDocumentation.pdf")
>>> toc = doc.get_toc(False) # we need the detailed TOC
>>> # list of level 1 indices and their titles
>>> lvl1 = [(i, item[1]) for i, item in enumerate(toc) if item[0] == 1]
>>> for i, title in lvl1:
    d = toc[i][3] # get the destination dict
    d["collapse"] = True # collapse items underneath
    if "Python" in title: # show the 'Python' chapter
        d["color"] = (1, 0, 0) # in red,
        d["bold"] = True # bold and
        d["italic"] = True # italic
        doc.set_toc_item(i, dest_dict=d) # update this toc item
>>> doc.save("NEWSWIG.pdf", garbage=3, deflate=True)
```

In the previous example, we have changed only 42 of the 1240 TOC items of the file.

can_save_incrementally()

- New in v1.16.0

Check whether the document can be saved incrementally. Use it to choose the right option without encountering exceptions.

scrub(*attached_files=True, clean_pages=True, embedded_files=True, hidden_text=True, javascript=True, metadata=True, redactions=True, redact_images=0, remove_links=True, reset_fields=True, reset_responses=True, thumbnails=True, xml_metadata=True*)

- New in v1.16.14

PDF only: Remove potentially sensitive data from the PDF. This function is inspired by the similar “Sanitize” function in Adobe Acrobat products. The process is configurable by a number of options, which are all *True* by default.

Parameters

- **attached_files** (*bool*) – Search for ‘FileAttachment’ annotations and remove the file content.

- **clean_pages** (*bool*) – Remove any comments from page painting sources. If this option is set to *False*, then this is also done for *hidden_text* and *redactions*.
- **embedded_files** (*bool*) – Remove embedded files.
- **hidden_text** (*bool*) – Remove OCR'd text and invisible text⁷.
- **javascript** (*bool*) – Remove JavaScript sources.
- **metadata** (*bool*) – Remove PDF standard metadata.
- **redactions** (*bool*) – Apply redaction annotations.
- **redact_images** (*int*) – how to handle images if applying redactions. One of 0 (ignore), 1 (blank out overlaps) or 2 (remove).
- **remove_links** (*bool*) – Remove all links.
- **reset_fields** (*bool*) – Reset all form fields to their defaults.
- **reset_responses** (*bool*) – Remove all responses from all annotations.
- **thumbnails** (*bool*) – Remove thumbnail images from pages.
- **xml_metadata** (*bool*) – Remove XML metadata.

save(*outfile*, *garbage*=0, *clean*=False, *deflate*=False, *deflate_images*=False, *deflate_fonts*=False, *incremental*=False, *ascii*=False, *expand*=0, *linear*=False, *pretty*=False, *no_new_id*=False, *encryption*=PDF_ENCRYPT_NONE, *permissions*=-1, *owner_pw*=None, *user_pw*=None)

- Changed in v1.18.7
- Changed in v1.19.0

PDF only: Saves the document in its **current state**.

Parameters

- **outfile** (*str*, *Path*, *fp*) – The file path, `pathlib.Path` or file object to save to. A file object must have been created before via `open(...)` or `io.BytesIO()`. Choosing `io.BytesIO()` is similar to `Document.tobytes()` below, which equals the `getvalue()` output of an internally created `io.BytesIO()`.
- **garbage** (*int*) – Do garbage collection. Positive values exclude “incremental”.
 - 0 = none
 - 1 = remove unused (unreferenced) objects.
 - 2 = in addition to 1, compact the *xref* table.
 - 3 = in addition to 2, merge duplicate objects.
 - 4 = in addition to 3, check *stream* objects for duplication. This may be slow because such data are typically large.
- **clean** (*bool*) – Clean and sanitize content streams¹. Corresponds to “mu-tool clean -sc”.

⁷ This only works under certain conditions. For example, if there is normal text covered by some image on top of it, then this is undetectable and the respective text is **not** removed. Similar is true for white text on white background, and so on.

¹ Content streams describe what (e.g. text or images) appears where and how on a page. PDF uses a specialized mini language similar to PostScript to do this (pp. 643 in *Adobe PDF References*), which gets interpreted when a page is loaded.

- **deflate** (*bool*) – Deflate (compress) uncompressed streams.
- **deflate_images** (*bool*) – (*new in v1.18.3*) Deflate (compress) uncompressed image streams⁴.
- **deflate_fonts** (*bool*) – (*new in v1.18.3*) Deflate (compress) uncompressed fontfile streams^{Page 180, 4}.
- **incremental** (*bool*) – Only save changes to the PDF. Excludes “garbage” and “linear”. Can only be used if *outfile* is a string or a `pathlib.Path` and equal to `Document.name`. Cannot be used for files that are decrypted or repaired and also in some other cases. To be sure, check `Document.can_save_incrementally()`. If this is false, saving to a new file is required.
- **ascii** (*bool*) – convert binary data to ASCII.
- **expand** (*int*) – Decompress objects. Generates versions that can be better read by some other programs and will lead to larger files.
 - 0 = none
 - 1 = images
 - 2 = fonts
 - 255 = all
- **linear** (*bool*) – Save a linearised version of the document. This option creates a file format for improved performance for Internet access. Excludes “incremental”.
- **pretty** (*bool*) – Prettify the document source for better readability. PDF objects will be reformatted to look like the default output of `Document.xref_object()`.
- **no_new_id** (*bool*) – Suppress the update of the file’s /ID field. If the file happens to have no such field at all, also suppress creation of a new one. Default is `False`, so every save will lead to an updated file identification.
- **permissions** (*int*) – (*new in v1.16.0*) Set the desired permission levels. See [Document Permissions](#) for possible values. Default is granting all.
- **encryption** (*int*) – (*new in v1.16.0*) set the desired encryption method. See [PDF encryption method codes](#) for possible values.
- **owner_pw** (*str*) – (*new in v1.16.0*) set the document’s owner password. (*Changed in v1.18.3*) If not provided, the user password is taken if provided. The string length must not exceed 40 characters.
- **user_pw** (*str*) – (*new in v1.16.0*) set the document’s user password. The string length must not exceed 40 characters.

Note: The method does not check, whether a file of that name already exists, will hence not ask for confirmation, and overwrite the file. It is your responsibility as a programmer to handle this.

⁴ These parameters cause separate handling of stream categories: use it together with `expand` to restrict decompression to streams other than images / fontfiles.

ez_save(*args, **kwargs)

- New in v1.18.11

PDF only: The same as [Document.save\(\)](#) but with the changed defaults *deflate=True*, *garbage=3*.

saveIncr()

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

Note: Saving incrementally may be required if the document contains verified signatures which would be invalidated by saving to a new file.

tobytes(garbage=0, clean=False, deflate=False, deflate_images=False, deflate_fonts=False, ascii=False, expand=0, linear=False, pretty=False, no_new_id=False, encryption=PDF_ENCRYPT_NONE, permissions=-1, owner_pw=None, user_pw=None)

- Changed in v1.18.7
- Changed in v1.19.0

PDF only: Writes the **current content of the document** to a bytes object instead of to a file. Obviously, you should be wary about memory requirements. The meanings of the parameters exactly equal those in [save\(\)](#). Chapter FAQ contains an example for using this method as a pre-processor to [pdfwr](#).

(Changed in v1.16.0) for extended encryption support.

Return type

bytes

Returns

a bytes object containing the complete document.

search_page_for(pno, text, quads=False)

Search for “text” on page number “pno”. Works exactly like the corresponding [Page.search_for\(\)](#). Any integer $-\infty < \text{pno} < \text{page_count}$ is acceptable.

insert_pdf(docsrc, from_page=-1, to_page=-1, start_at=-1, rotate=-1, links=True, annots=True, show_progress=0, final=1)

- Changed in v1.19.3 - as a fix to issue [#537](#), form fields are always excluded.

PDF only: Copy the page range [**from_page**, **to_page**] (including both) of PDF document *docsrc* into the current one. Inserts will start with page number *start_at*. Value -1 indicates default values. All pages thus copied will be rotated as specified. Links and annotations can be excluded in the target, see below. All page numbers are 0-based.

Parameters

- **docsrc** (*Document*) – An opened PDF *Document* which must not be the current document. However, it may refer to the same underlying file.
- **from_page** (*int*) – First page number in *docsrc*. Default is zero.
- **to_page** (*int*) – Last page number in *docsrc* to copy. Defaults to last page.

- **start_at** (*int*) – First copied page, will become page number *start_at* in the target. Default -1 appends the page range to the end. If zero, the page range will be inserted before current first page.
- **rotate** (*int*) – All copied pages will be rotated by the provided value (degrees, integer multiple of 90).
- **links** (*bool*) – Choose whether (internal and external) links should be included in the copy. Default is *True*. Internal links to outside the copied page range are **always excluded**.
- **annots** (*bool*) – (*new in v1.16.1*) choose whether annotations should be included in the copy. (*Fixed in v1.19.3*) Form fields can never be copied.
- **show_progress** (*int*) – (*new in v1.17.7*) specify an interval size greater zero to see progress messages on `sys.stdout`. After each interval, a message like `Inserted 30 of 47 pages.` will be printed.
- **final** (*int*) – (*new in v1.18.0*) controls whether the list of already copied objects should be **dropped** after this method, default *True*. Set it to 0 except for the last one of multiple insertions from the same source PDF. This saves target file size and speeds up execution considerably.

Note:

1. If *from_page* > *to_page*, pages will be **copied in reverse order**. If $0 \leq \text{from_page} == \text{to_page}$, then one page will be copied.
 2. *docsrc* TOC entries **will not be copied**. It is easy however, to recover a table of contents for the resulting document. Look at the examples below and at program [PDFjoiner.py](#) in the *examples* directory: it can join PDF documents and at the same time piece together respective parts of the tables of contents.
-

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

PDF only: Insert an empty page.

Parameters

- **pno** (*int*) – page number in front of which the new page should be inserted. Must be in $1 < \text{pno} \leq \text{page_count}$. Special values -1 and *doc.page_count* insert **after** the last page.
- **width** (*float*) – page width.
- **height** (*float*) – page height.

Return type

Page

Returns

the created page object.

insert_page(*pno*, *text*=None, *fontsize*=11, *width*=595, *height*=842, *fontname*='helv', *fontfile*=None, *color*=None)

PDF only: Insert a new page and insert some text. Convenience function which combines [Document.new_page\(\)](#) and (parts of) [Page.insert_text\(\)](#).

Parameters

pno (*int*) – page number (0-based) **in front of which** to insert. Must

be in `range(-1, doc.page_count + 1)`. Special values `-1` and `doc.page_count` insert **after** the last page.

Changed in v1.14.12

This is now a positional parameter

For the other parameters, please consult the aforementioned methods.

Return type

int

Returns

the result of `Page.insert_text()` (number of successfully inserted lines).

`delete_page(pno=-1)`

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

- Changed in v1.18.14: support Python's `del` statement.

Parameters

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

`delete_pages(*args, **kws)`

- Changed in v1.18.13: more flexibility specifying pages to delete.
- Changed in v1.18.14: support Python's `del` statement.

PDF only: Delete multiple pages given as 0-based numbers.

Format 1: Use keywords. Represents the old format. A contiguous range of pages is removed.

- “from_page”: first page to delete. Zero if omitted.
- “to_page”: last page to delete. Last page in document if omitted. Must not be less than “from_page”.

Format 2: Two page numbers as positional parameters. Handled like Format 1.

Format 3: One positional integer parameter. Equivalent to `Page.delete_page()`.

Format 4: One positional parameter of type *list*, *tuple* or *range()* of page numbers. The items of this sequence may be in any order and may contain duplicates.

Format 5: (*New in v1.18.14*) Using the Python `del` statement and index / slice notation is now possible.

Note: (*Changed in v1.14.17, optimized in v1.17.7*) In an effort to maintain a valid PDF structure, this method and `delete_page()` will also deactivate items in the table of contents which point to deleted pages. “Deactivation” here means, that the bookmark will point to nowhere and the title will be shown grayed-out by supporting PDF viewers. The overall TOC structure is left intact.

It will also remove any **links on remaining pages** which point to a deleted one. This action may have an extended response time for documents with many pages.

Following examples will all delete pages 500 through 519:

- `doc.delete_pages(500, 519)`
- `doc.delete_pages(from_page=500, to_page=519)`

- `doc.delete_pages((500, 501, 502, ... , 519))`
- `doc.delete_pages(range(500, 520))`
- `del doc[500:520]`
- `del doc[(500, 501, 502, ... , 519)]`
- `del doc[range(500, 520)]`

For the *Adobe PDF References* the above takes about 0.6 seconds, because the remaining 1290 pages must be cleaned from invalid links.

In general, the performance of this method is dependent on the number of remaining pages – **not** on the number of deleted pages: in the above example, **deleting all pages except** those 20, will need much less time.

`copy_page(pno, to=-1)`

PDF only: Copy a page reference within the document.

Parameters

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

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

`fullcopy_page(pno, to=-1)`

- New in v1.14.17

PDF only: Make a full copy (duplicate) of a page.

Parameters

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

Note:

- In contrast to *copy_page()*, this method creates a new page object (with a new *xref*), which can be changed independently from the original.
 - Any Popup and “IRT” (“in response to”) annotations are **not copied** to avoid potentially incorrect situations.
-

`move_page(pno, to=-1)`

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

Parameters

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

need_appearances(*value=None*)

- New in v1.17.4

PDF only: Get or set the */NeedAppearances* property of Form PDFs. Quote: “(Optional) A flag specifying whether to construct appearance streams and appearance dictionaries for all widget annotations in the document ... Default value: false.” This may help controlling the behavior of some readers / viewers.

Parameters

value (*bool*) – set the property to this value. If omitted or *None*, inquire the current value.

Return type

bool

Returns

- None: not a Form PDF, or property not defined.
- True / False: the value of the property (either just set or existing for inquiries). Has no effect if no Form PDF.

get_sigflags()

PDF only: Return whether the document contains signature fields. This is an optional PDF property: if not present (return value -1), no conclusions can be drawn – the PDF creator may just not have bothered using it.

Return type

int

Returns

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

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

- Changed in v1.14.16: The sequence of positional parameters “name” and “buffer” has been changed to comply with the call pattern of other functions.

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

- **name** (*str*) – entry identifier, **must not already exist**.
- **buffer** (*bytes, bytearray, BytesIO*) – file contents.
(Changed in v1.14.13) *io.BytesIO* is now also supported.
- **filename** (*str*) – optional filename. Documentation only, will be set to *name* if *None*.

- **ufilename** (*str*) – optional unicode filename. Documentation only, will be set to *filename* if *None*.
- **desc** (*str*) – optional description. Documentation only, will be set to *name* if *None*.

Return type

int

Returns

(*Changed in v1.18.13*) The method now returns the [xref](#) of the inserted file. In addition, the file object now will be automatically given the PDF keys / `CreationDate` and / `ModDate` based on the current date-time.

embfile_count()

- Changed in v1.14.16: This is now a method. In previous versions, this was a property.

PDF only: Return the number of embedded files.

embfile_get(*item*)

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

Parameters

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

Return type

bytes

embfile_del(*item*)

- Changed in v1.14.16: Items can now be deleted by index, too.

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

Parameters

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

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

embfile_info(*item*)

- Changed in v1.18.13

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

Parameters

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

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
- *creationDate* – (*New in v1.18.13*) (*str*) date-time of item creation in PDF format
- *modDate* – (*New in v1.18.13*) (*str*) date-time of last change in PDF format
- *collection* – (*New in v1.18.13*) (*int*) [xref](#) of the associated PDF portfolio item if any, else zero.
- *checksum* – (*New in v1.18.13*) (*str*) a hashcode of the stored file content as a hexadecimal string. Should be MD5 according to PDF specifications, but be prepared to see other hashing algorithms.

embfile_names()

- New in v1.14.16

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

Return type

list

embfile_upd(*item*, *buffer*=None, *filename*=None, *ufilename*=None, *desc*=None)

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

Parameters

- **item** (*int*/*str*) – index or name of entry. An integer must be in `range(embfile_count())`.
- **buffer** (*bytes*, *bytearray*, *BytesIO*) – the new file content.
(*Changed in v1.14.13*) *io.BytesIO* is now also supported.
- **filename** (*str*) – the new filename.
- **ufilename** (*str*) – the new unicode filename.
- **desc** (*str*) – the new description.

(*Changed in v1.18.13*) The method now returns the [xref](#) of the file object.

Return type

int

Returns

xref of the file object. Automatically, its `/ModDate` PDF key will be updated with the current date-time.

close()

Release objects and space allocations associated with the document. If created from a file, also closes *filename* (releasing control to the OS). Explicitly closing a document is equivalent to deleting it, `del doc`, or assigning it to something else like `doc = None`.

xref_object(xref, compressed=False, ascii=False)

- New in v1.16.8
- Changed in v1.18.10

PDF only: Return the definition source of a PDF object.

Parameters

- **xref** (*int*) – the object’s `:data`xref``. *Changed in v1.18.10:* A value of -1 returns the PDF trailer source.
- **compressed** (*bool*) – whether to generate a compact output with no line breaks or spaces.
- **ascii** (*bool*) – whether to ASCII-encode binary data.

Return type

str

Returns

The object definition source.

pdf_catalog()

- New in v1.16.8

PDF only: Return the *xref* number of the PDF catalog (or root) object. Use that number with *Document.xref_object()* to see its source.

pdf_trailer(compressed=False)

- New in v1.16.8

PDF only: Return the trailer source of the PDF, which is usually located at the PDF file’s end. This is *Document.xref_object()* with an *xref* argument of -1.

xref_stream(xref)

- New in v1.16.8

PDF only: Return the **decompressed** contents of the *xref* stream object.

Parameters

xref (*int*) – *xref* number.

Return type

bytes

Returns

the (decompressed) stream of the object.

xref_stream_raw(xref)

- New in v1.16.8

PDF only: Return the **unmodified** (esp. **not decompressed**) contents of the *xref* stream object. Otherwise equal to *Document.xref_stream()*.

Return type

bytes

Returns

the (original, unmodified) stream of the object.

update_object(*xref*, *obj_str*, *page=None*)

- New in v1.16.8

PDF only: Replace object definition of *xref* with the provided string. The *xref* may also be new, in which case this instruction completes the object definition. If a page object is also given, its links and annotations will be reloaded afterwards.

Parameters

- **xref** (*int*) – *xref* number.
- **obj_str** (*str*) – a string containing a valid PDF object definition.
- **page** (*Page*) – a page object. If provided, indicates, that annotations of this page should be refreshed (reloaded) to reflect changes incurred with links and / or annotations.

Return type

int

Returns

zero if successful, otherwise an exception will be raised.

update_stream(*xref*, *data*, *new=False*, *compress=True*)

- New in v1.16.8
- Changed in v1.19.2: added parameter “compress”
- Changed in v1.19.6: deprecated parameter “new”. Now confirms that the object is a PDF dictionary object.

Replace the stream of an object identified by *xref*, which must be a PDF dictionary. If the object is no *stream*, it will be turned into one. The function automatically performs a compress operation (“deflate”) where beneficial.

Parameters

- **xref** (*int*) – *xref* number.
- **stream** (*bytes*/*bytearray*/*BytesIO*) – the new content of the stream.
(*Changed in v1.14.13:*) *io.BytesIO* objects are now also supported.
- **new** (*bool*) – *deprecated* and ignored. Will be removed some time after v1.20.0.
- **compress** (*bool*) – whether to compress the inserted stream. If True (default), the stream will be inserted using /FlateDecode compression (if beneficial), otherwise the stream will inserted as is.

Raises

ValueError – if *xref* does not represent a PDF dict. An empty dictionary <<>> is accepted. So if you just created the *xref* and want to give it a stream, first execute `doc.update_object(xref, "<<>>")`, and then insert the stream data with this method.

The method is primarily (but not exclusively) intended to manipulate streams containing PDF operator syntax (see pp. 643 of the *Adobe PDF References*) as it is the case for e.g. page content streams.

If you update a contents stream, consider using save parameter *clean=True* to ensure consistency between PDF operator source and the object structure.

Example: Let us assume that you no longer want a certain image appear on a page. This can be achieved by deleting the respective reference in its contents source(s) – and indeed: the image will be gone after reloading the page. But the page’s *resources* object would still show the image as being referenced by the page. This save option will clean up any such mismatches.

xref_copy(*source*, *target*, *, *keep=None*)

- New in v1.19.5

PDF Only: Make *target* xref an exact copy of *source*. If *source* is a *stream*, then these data are also copied.

Parameters

- **source** (*int*) – the source *xref*. It must be an existing **dictionary** object.
- **target** (*int*) – the target xref. Must be an existing **dictionary** object. If the xref has just been created, make sure to initialize it as a PDF dictionary with the minimum specification <<>>.
- **keep** (*list*) – an optional list of top-level keys in *target*, that should not be removed in preparation of the copy process.

Note:

- This method has much in common with Python’s *dict* method *copy()*.
 - Both xref numbers must represent existing dictionaries.
 - Before data is copied from *source*, all *target* dictionary keys are deleted. You can specify exceptions from this in the *keep* list. If *source* however has a same-named key, its value will still replace the target.
 - If *source* is a *stream* object, then these data will also be copied over, and *target* will be converted to a stream object.
 - A typical use case is to replace or remove an existing image without using redaction annotations. Example scripts can be seen [here](#).
-

extract_image(*xref*)

PDF Only: Extract data and meta information of an image stored in the document. The output can directly be used to be stored as an image file, as input for PIL, *Pixmap* creation, etc. This method avoids using pixmaps wherever possible to present the image in its original format (e.g. as JPEG).

Parameters

xref (*int*) – *xref* of an image object. If this is not in *range(1, doc.xref_length())*, or the object is no image or other errors occur, *None* is returned and no exception is raised.

Return type

dict

Returns

a dictionary with the following keys

- *ext* (*str*) image type (e.g. *'jpeg'*), usable as image file extension
- *smask* (*int*) [xref](#) number of a stencil (/SMask) image or zero
- *width* (*int*) image width
- *height* (*int*) image height
- *colorspace* (*int*) the image's *colorspace.n* number.
- *cs-name* (*str*) the image's *colorspace.name*.
- *xres* (*int*) resolution in x direction. Please also see [resolution](#).
- *yres* (*int*) resolution in y direction. Please also see [resolution](#).
- *image* (*bytes*) image data, usable as image file content

```
>>> d = doc.extract_image(1373)
>>> d
{'ext': 'png', 'smask': 2934, 'width': 5, 'height': 629, 'colorspace': 3, 'xres
→': 96,
'yres': 96, 'cs-name': 'DeviceRGB',
'image': b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x00\x05\ ...'}
>>> imgout = open(f"image.{d['ext']}", "wb")
>>> imgout.write(d["image"])
102
>>> imgout.close()
```

Note: There is a functional overlap with `pix = fitz.Pixmap(doc, xref)`, followed by a `pix.tobytes()`. Main differences are that `extract_image`, (1) does not always deliver PNG image formats, (2) is **very** much faster with non-PNG images, (3) usually results in much less disk storage for extracted images, (4) returns `None` in error cases (generates no exception). Look at the following example images within the same PDF.

- xref 1268 is a PNG – Comparable execution time and identical output:

```
In [23]: %timeit pix = fitz.Pixmap(doc, 1268);pix.tobytes()
10.8 ms ± 52.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
In [24]: len(pix.tobytes())
Out[24]: 21462

In [25]: %timeit img = doc.extract_image(1268)
10.8 ms ± 86 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
In [26]: len(img["image"])
Out[26]: 21462
```

- xref 1186 is a JPEG – `Document.extract_image()` is **many times faster** and produces a **much smaller** output (2.48 MB vs. 0.35 MB):

```
In [27]: %timeit pix = fitz.Pixmap(doc, 1186);pix.tobytes()
341 ms ± 2.86 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [28]: len(pix.tobytes())
Out[28]: 2599433

In [29]: %timeit img = doc.extract_image(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
```

extract_font(*xref*, *info_only=False*, *named=None*)

- Changed in v1.19.4: return a dictionary if `named == True`.

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](#)).

arg int xref

PDF object number of the font to extract.

arg bool info_only

only return font information, not the buffer. To be used for information-only purposes, avoids allocation of large buffer areas.

arg bool named

If true, a dictionary with the following keys is returned: 'name' (font base name), 'ext' (font file extension), 'type' (font type), 'content' (font file content).

rtype

tuple, dict

returns

a tuple (`basename`, `ext`, `type`, `content`), where `ext` is a 3-byte suggested file extension (*str*), `basename` is the font's name (*str*), `type` is the font's type (e.g. "Type1") and `content` 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 not embedded and thus cannot be extracted. This is the case for e.g. the [PDF Base 14 Fonts](#) and Type 3 fonts.

Example:

```
>>> # store font as an external file
>>> name, ext, _, content = doc.extract_font(4711)
>>> # assuming content is not None:
>>> ofile = open(name + "." + ext, "wb")
>>> ofile.write(content)
>>> ofile.close()
```

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

Note:

- The returned `basename` in general is **not** the original file name, but it probably has some similarity.
- If parameter `named == True`, a dictionary with the following keys is returned: `{'name': 'T1', 'ext': 'n/a', 'type': 'Type3', 'content': b''}`.

xref_xml_metadata()

- New in v1.16.8

PDF only: Return the *xref* of the document's XML metadata.

has_links()**has_annots()**

- New in v1.18.7

PDF only: Check whether there are links, resp. annotations anywhere in the document.

Returns

True / False. As opposed to fields, which are also stored in a central place of a PDF document, the existence of links / annotations can only be detected by parsing each page. These methods are tuned to do this efficiently and will immediately return, if the answer is *True* for a page. For PDFs with many thousand pages however, an answer may take some time⁶ if no link, resp. no annotation is found.

subset_fonts()

- New in v1.18.7, changed in v1.18.9

PDF only: Investigate eligible fonts for their use by text in the document. If a font is supported and a size reduction is possible, that font is replaced by a version with a character subset.

Use this method immediately before saving the document. The following features and restrictions apply for the time being:

- Package **fontTools** **must be installed**. It is required for creating the font subsets. If not installed, the method raises an `ImportError` exception.
- Supported font types only include embedded OTF, TTF and WOFF that are **not already subsets**.
- **Changed in v1.18.9:** A subset font directly replaces its original – text remains untouched and **is not rewritten**. It thus should retain all its properties, like spacing, hiddenness, control by Optional Content, etc.

The greatest benefit can be achieved when creating new PDFs using large fonts like is typical for Asian scripts. In these cases, the set of actually used unicodes mostly is small compared to the number of glyphs in the font. Using this feature can easily reduce the embedded font binary by two orders of magnitude – from several megabytes to a low two-digit kilobyte amount.

journal_enable()

- New in v1.19.0

PDF only: Enable journalling. Use this before you start logging operations.

journal_start_op(name)

- New in v1.19.0

PDF only: Start journalling an “*operation*” identified by a string “*name*”. Updates will fail for a journal-enabled PDF, if no operation has been started.

⁶ For a *False* the **complete document** must be scanned. Both methods **do not load pages**, but only scan object definitions. This makes them at least 10 times faster than application-level loops (where total response time roughly equals the time for loading all pages). For the *Adobe PDF References* (756 pages) and the Pandas documentation (over 3070 pages) – both have no annotations – the method needs about 11 ms for the answer *False*. So response times will probably become significant only well beyond this order of magnitude.

journal_stop_op()

- New in v1.19.0

PDF only: Stop the current operation. The updates between start and stop of an operation belong to the same unit of work and will be undone / redone together.

journal_position()

- New in v1.19.0

PDF only: Return the numbers of the current operation and the total operation count.

Returns

a tuple (**step**, **steps**) containing the current operation number and the total number of operations in the journal. If **step** is 0, we are at the top of the journal. If **step** equals **steps**, we are at the bottom. Updating the PDF with anything other than undo or redo will automatically remove all journal entries after the current one and the new update will become the new last entry in the journal. The updates corresponding to the removed journal entries will be permanently lost.

journal_op_name(*step*)

- New in v1.19.0

PDF only: Return the name of operation number *step*.

journal_can_do()

- New in v1.19.0

PDF only: Show whether forward (“redo”) and / or backward (“undo”) executions are possible from the current journal position.

Returns

a dictionary {"undo": bool, "redo": bool}. The respective method is available if its value is True.

journal_undo()

- New in v1.19.0

PDF only: Revert (undo) the current step in the journal. This moves towards the journal’s top.

journal_redo()

- New in v1.19.0

PDF only: Re-apply (redo) the current step in the journal. This moves towards the journal’s bottom.

journal_save(*filename*)

- New in v1.19.0

PDF only: Save the journal to a file.

Parameters

filename (*str*, *fp*) – either a filename as string or a file object opened as “wb” (or an `io.BytesIO()` object).

journal_load(*filename*)

- New in v1.19.0

PDF only: Load journal from a file. Enables journalling for the document. If journalling is already enabled, an exception is raised.

Parameters

filename (*str*, *fp*) – the filename (*str*) of the journal or a file object opened as “rb” (or an `io.BytesIO()` object).

save_snapshot()

- New in v1.19.0

PDF only: Saves a “snapshot” of the document. This is a PDF document with a special, incremental-save format compatible with journalling – therefore no save options are available. Saving a snapshot is not possible for new documents.

This is a normal PDF document with no usage restrictions whatsoever. If it is not being changed in any way, it can be used together with its journal to undo / redo operations or continue updating.

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](#)

is_closed

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.load_page\(\)](#)) 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

is_dirty

True if this is a PDF document and contains unsaved changes, else *False*.

Type

bool

is_pdf

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

Type

bool

is_form_pdf

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

(*Changed in v1.16.4*) Returns the total number of (root) form fields.

Type

bool,int

is_reflowable

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

is_repaired

- New in v1.18.2

True if PDF has been repaired during open (because of major structure issues). Always *False* for non-PDF documents. If true, more details have been stored in `TOOLS.mupdf_warnings()`, and `Document.can_save_incrementally()` will return *False*.

Type

bool

needs_pass

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

Type

bool

is_encrypted

This indicator initially equals `Document.needs_pass`. After successful authentication, it is set to *False* to reflect the situation.

Type

bool

permissions

- Changed in v1.16.0: This is now an integer comprised of bit indicators. Was a dictionary previously.

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

Type

int

metadata

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

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

- *format* contains the document format (e.g. 'PDF-1.6', 'XPS', 'EPUB').
- *encryption* either contains *None* (no encryption), or a string naming an encryption method (e.g. 'Standard V4 R4 128-bit RC4'). Note that an encryption method may be specified **even if** `needs_pass=False`. In such cases not all permissions will probably have been granted. Check [Document.permissions](#) for details.
- If the date fields contain valid data (which need not be the case at all!), they are strings in the PDF-specific timestamp format "D:<TS><TZ>", where
 - <TS> is the 12 character ISO timestamp *YYYYMMDDhhmmss* (*YYYY* - year, *MM* - month, *DD* - day, *hh* - hour, *mm* - minute, *ss* - second), and

- <TZ> is a time zone value (time intervall relative to GMT) containing a sign ('+' or '-'), the hour (*hh*), and the minute ('*mm*', note the apostrophies!).
- A Paraguayan value might hence look like *D:20150415131602-04'00'*, which corresponds to the timestamp April 15, 2015, at 1:16:02 pm local time Asuncion.

Type

dict

name

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

Type

str

page_count

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

chapter_count

- New in v1.17.0

Contains the number of chapters in the document. Always at least 1. Relevant only for document types with chapter support (EPUB currently). Other documents will return 1.

Type

int

last_location

- New in v1.17.0

Contains (chapter, pno) of the document's last page. Relevant only for document types with chapter support (EPUB currently). Other documents will return (0, `page_count - 1`) and (0, -1) if it has no pages.

Type

int

FormFonts

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

Type

list

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

6.5.1 set_metadata() 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.set_metadata({})  # clear all fields
>>> doc.metadata          # look again to show what happened
{'producer': 'none', 'format': 'PDF 1.4', 'encryption': None, 'author': 'none',
'modDate': 'none', 'keywords': 'none', 'title': 'none', 'creationDate': 'none',
'creator': 'none', 'subject': 'none'}
>>> doc._delXmlMetadata() # clear any XML metadata
>>> doc.save("anonymous.pdf", garbage = 4)      # save anonymized doc
```

6.5.2 set_toc() Demonstration

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

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

6.5.3 insert_pdf() 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.get_toc(False)               # save TOC 1
>>> toc2 = doc2.get_toc(False)               # save TOC 2
>>> doc1.insert_pdf(doc2)                    # doc2 at end of doc1
>>> for t in toc2:                           # increase toc2 page numbers
    t[2] += pages1                          # by old len(doc1)
>>> doc1.set_toc(toc1 + toc2)                # now result has total TOC
```

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

(2) More examples:

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

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

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

6.5.4 Other Examples

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

```
for i in range(doc.page_count):
    imglist = doc.get_page_images(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.save("p%s-%s.png" % (i, xref))
        else:                               # CMYK: must convert first
            pix0 = fitz.Pixmap(fitz.csRGB, pix)
            pix0.save("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.set_rotation(90)
```

6.6 DocumentWriter

- New in v1.21.0

This class represents a utility which can output various document types supported by MuPDF.

In PyMuPDF only used for outputting PDF documents whose pages are populated by *Story* DOMs.

Using *DocumentWriter* also for other document types might happen in the future.

Method / Attribute	Short Description
<i>DocumentWriter.begin_page()</i>	start a new output page
<i>DocumentWriter.end_page()</i>	finish the current output page
<i>DocumentWriter.close()</i>	flush pending output and close the file

Class API

class DocumentWriter

__init__(*self*, *path*, *options=None*)

Create a document writer object, passing a Python file pointer or a file path. Options to use when saving the file may also be passed.

Parameters

- **path** – the output file. This may be a string file name, or any Python file pointer.

Note: By using a `io.BytesIO()` object as file pointer, a document writer can create a PDF in memory. Subsequently, this PDF can be re-opened for input and be further manipulated. This technique is used by several example scripts in *Stories recipes*.

- **options** (*str*) – specify saving options for the output PDF. Typical are “compress” or “clean”. More possible values may be taken from help output of the *mutool convert* CLI utility.

begin_page(*mediabox*)

Start a new output page of a given dimension.

Parameters

mediabox (*rect_like*) – a rectangle specifying the page size. After this method, output operations may write content to the page.

end_page()

Finish a page. This flushes any pending data and appends the page to the output document.

close()

Close the output file. This method is required for writing any pending data.

For usage examples consult the section of *Story*.

6.7 Font

- New in v1.16.18

This class represents a font as defined in MuPDF (*fz_font_s* structure). It is required for the new class *TextWriter* and the new *Page.write_text()*. Currently, it has no connection to how fonts are used in methods *Page.insert_text()* or *Page.insert_textbox()*, respectively.

A Font object also contains useful general information, like the font bbox, the number of defined glyphs, glyph names or the bbox of a single glyph.

Method / Attribute	Short Description
<i>glyph_advance()</i>	Width of a character
<i>glyph_bbox()</i>	Glyph rectangle
<i>glyph_name_to_unicode()</i>	Get unicode from glyph name
<i>has_glyph()</i>	Return glyph id of unicode
<i>text_length()</i>	Compute string length
<i>char_lengths()</i>	Tuple of char widths of a string
<i>unicode_to_glyph_name()</i>	Get glyph name of a unicode
<i>valid_codepoints()</i>	Array of supported unicodes
<i>ascender</i>	Font ascender
<i>descender</i>	Font descender
<i>bbox</i>	Font rectangle
<i>buffer</i>	Copy of the font's binary image
<i>flags</i>	Collection of font properties
<i>glyph_count</i>	Number of supported glyphs
<i>name</i>	Name of font
<i>is_writable</i>	Font usable with <i>TextWriter</i>

Class API

class Font

```
__init__(self, fontname=None, fontfile=None,
fontbuffer=None, script=0, language=None, ordering=-1, is_bold=0,
is_italic=0, is_serif=0)
```

Font constructor. The large number of parameters are used to locate font, which most closely resembles the requirements. Not all parameters are ever required – see the below pseudo code explaining the logic how the parameters are evaluated.

Parameters

- **fontname** (*str*) – one of the *PDF Base 14 Fonts* or CJK fontnames. Also possible are a select few other names like (watch the correct spelling): “Arial”, “Times”, “Times Roman”.

(Changed in v1.17.5)

If you have installed *pymupdf-fonts*, there are also new “reserved” fontnames available, which are listed in *fitz_fonts* and in the table further down.

- **fontfile** (*str*) – the filename of a fontfile somewhere on your system¹.

¹ MuPDF does not support all fontfiles with this feature and will raise exceptions like “mupdf: FT_New_Memory_Face((null)): unknown file format”, if it encounters issues. The *TextWriter* methods check *Font.is_writable*.

- **fontbuffer** (*bytes, bytearray, io.BytesIO*) – a fontfile loaded in memory^{Page 201, 1}.
- **script** (*in*) – the number of a UCDN script. Currently supported in PyMuPDF are numbers 24, and 32 through 35.
- **language** (*str*) – one of the values “zh-Hant” (traditional Chinese), “zh-Hans” (simplified Chinese), “ja” (Japanese) and “ko” (Korean). Otherwise, all ISO 639 codes from the subsets 1, 2, 3 and 5 are also possible, but are currently documentary only.
- **ordering** (*int*) – an alternative selector for one of the CJK fonts.
- **is_bold** (*bool*) – look for a bold font.
- **is_italic** (*bool*) – look for an italic font.
- **is_serif** (*bool*) – look for a serified font.

Returns

a MuPDF font if successful. This is the overall sequence of checks to determine an appropriate font:

Argument	Action
fontfile?	Create font from file, exception if failure.
font-buffer?	Create font from buffer, exception if failure.
ordering>=0	Create universal font, always succeeds.
font-name?	Create a Base-14 font, universal font, or font provided by pymupdf-fonts . See table below.

Note: With the usual reserved names “helv”, “tiro”, etc., you will create fonts with the expected names “Helvetica”, “Times-Roman” and so on. **However**, and in contrast to [Page.insert_font\(\)](#) and friends,

- a font file will **always** be embedded in your PDF,
- Greek and Cyrillic characters are supported without needing the *encoding* parameter.

Using *ordering* `>= 0`, or fontnames “cjk”, “china-t”, “china-s”, “japan” or “korea” will **always create the same “universal” font “Droid Sans Fallback Regular”**. This font supports **all Chinese, Japanese, Korean and Latin characters**, including Greek and Cyrillic. This is a sans-serif font.

Actually, you would rarely ever need another sans-serif font than **“Droid Sans Fallback Regular”**. **Except** that this font file is relatively large and adds about 1.65 MB (compressed) to your PDF file size. If you do not need CJK support, stick with specifying “helv”, “tiro” etc., and you will get away with about 35 KB compressed.

If you **know** you have a mixture of CJK and Latin text, consider just using `Font("cjk")` because this supports everything and also significantly (by a factor of up to three) speeds up execution: MuPDF will always find any character in this single font and never needs to check fallbacks.

But if you do use some other font, you will still automatically be able to also write CJK characters: MuPDF detects this situation and silently falls back to the universal font (which will then of course also be embedded in your PDF).

(*New in v1.17.5*) Optionally, some new “reserved” fontname codes become available if you install `pymupdf-fonts`, `pip install pymupdf-fonts`. “**Fira Mono**” is a mono-spaced sans font set and **FiraGO** is another non-serifed “universal” font set which supports all Latin (including Cyrillic and Greek) plus Thai, Arabian, Hebrew and Devanagari – but none of the CJK languages. The size of a FiraGO font is only a quarter of the “Droid Sans Fallback” size (compressed 400 KB vs. 1.65 MB) – **and** it provides the weights bold, italic, bold-italic – which the universal font doesn’t.

“**Space Mono**” is another nice and small mono-spaced font from Google Fonts, which supports Latin Extended characters and comes with all 4 important weights.

The following table maps a fontname code to the corresponding font. For the current content of the package please see its documentation:

Code	Fontname	New in	Comment
figo	FiraGO Regular	v1.0.0	narrower than Helvetica
figbo	FiraGO Bold	v1.0.0	
figit	FiraGO Italic	v1.0.0	
figbi	FiraGO Bold Italic	v1.0.0	
fimo	Fira Mono Regular	v1.0.0	
fimbo	Fira Mono Bold	v1.0.0	
spacemo	Space Mono Regular	v1.0.1	
spacembo	Space Mono Bold	v1.0.1	
spacemit	Space Mono Italic	v1.0.1	
spacembi	Space Mono Bold-Italic	v1.0.1	
math	Noto Sans Math Regular	v1.0.2	math symbols
music	Noto Music Regular	v1.0.2	musical symbols
symbol1	Noto Sans Symbols Regular	v1.0.2	replacement for “symb”
symbol2	Noto Sans Symbols2 Regular	v1.0.2	extended symbol set
notos	Noto Sans Regular	v1.0.3	alternative to Helvetica
notosit	Noto Sans Italic	v1.0.3	
notosbo	Noto Sans Bold	v1.0.3	
notosbi	Noto Sans BoldItalic	v1.0.3	

has_glyph(*chr*, *language=None*, *script=0*, *fallback=False*)

Check whether the unicode *chr* exists in the font or (option) some fallback font. May be used to check whether any “TOFU” symbols will appear on output.

Parameters

- **chr** (*int*) – the unicode of the character (i.e. *ord()*).
- **language** (*str*) – the language – currently unused.
- **script** (*int*) – the UCDN script number.
- **fallback** (*bool*) – (*new in v1.17.5*) perform an extended search in fallback fonts or restrict to current font (default).

Returns

(*changed in 1.17.7*) the glyph number. Zero indicates no glyph found.

valid_codepoints()

- New in v1.17.5

Return an array of unicodes supported by this font.

Returns

an `array.array`² of length at most `Font.glyph_count`. I.e. `chr()` of every item in this array has a glyph in the font without using fallbacks. This is an example display of the supported glyphs:

```
>>> import fitz
>>> font = fitz.Font("math")
>>> vuc = font.valid_codepoints()
>>> for i in vuc:
    print("%04X %s (%s)" % (i, chr(i), font.unicode_to_glyph_
↪name(i)))
0000
000D (CR)
0020 (space)
0021 ! (exclam)
0022 " (quotedbl)
0023 # (numbersign)
0024 $ (dollar)
0025 % (percent)
...
00AC ¬ (logicalnot)
00B1 ± (plusminus)
...
21D0 (arrowdblleft)
21D1 (arrowdblup)
21D2 (arrowdblright)
21D3 (arrowdbldown)
21D4 (arrowdblboth)
...
221E ∞ (infinity)
...
```

Note: This method only returns meaningful data for fonts having a CMAP (character map, charmap, the `/ToUnicode` PDF key). Otherwise, this array will have length 1 and contain zero only.

glyph_advance(*chr*, *language=None*, *script=0*, *wmode=0*)

Calculate the “width” of the character’s glyph (visual representation).

Parameters

- **chr** (*int*) – the unicode number of the character. Use `ord()`, not the character itself. Again, this should normally work even if a character is not supported by that font, because fallback fonts will be checked where necessary.
- **wmode** (*int*) – write mode, 0 = horizontal, 1 = vertical.

The other parameters are not in use currently.

Returns

a float representing the glyph’s width relative to **fontsize 1**.

² The built-in module `array` has been chosen for its speed and its compact representation of values.

glyph_name_to_unicode(name)

Return the unicode value for a given glyph name. Use it in conjunction with `chr()` if you want to output e.g. a certain symbol.

Parameters

name (*str*) – The name of the glyph.

Returns

The unicode integer, or 65533 = 0xFFFFD if the name is unknown. Examples: `font.glyph_name_to_unicode("Sigma") = 931`, `font.glyph_name_to_unicode("sigma") = 963`. Refer to the [Adobe Glyph List](#) publication for a list of glyph names and their unicode numbers. Example:

```
>>> font = fitz.Font("helv")
>>> font.has_glyph(font.glyph_name_to_unicode("infinity"))
True
```

glyph_bbox(chr, language=None, script=0)

The glyph rectangle relative to fontsize 1.

Parameters

chr (*int*) – `ord()` of the character.

Returns

a *Rect*.

unicode_to_glyph_name(ch)

Show the name of the character's glyph.

Parameters

ch (*int*) – the unicode number of the character. Use `ord()`, not the character itself.

Returns

a string representing the glyph's name. E.g. `font.glyph_name(ord("#")) = "numbersign"`. For an invalid code "notfound" is returned.

Note: (*Changed in v1.18.0*) This method and `Font.glyph_name_to_unicode()` no longer depend on a font and instead retrieve information from the **Adobe Glyph List**. Also available as `fitz.unicode_to_glyph_name()` and resp. `fitz.glyph_name_to_unicode()`.

text_length(text, fontsize=11)

Calculate the length in points of a unicode string.

Note: There is a functional overlap with `get_text_length()` for Base-14 fonts only.

Parameters

- **text** (*str*) – a text string, UTF-8 encoded.
- **fontsize** (*float*) – the fontsize.

Return type

float

Returns

the length of the string in points when stored in the PDF. If a character is not contained in the font, it will automatically be looked up in a fallback font.

Note: This method was originally implemented in Python, based on calling `Font.glyph_advance()`. For performance reasons, it has been rewritten in C for v1.18.14. To compute the width of a single character, you can now use either of the following without performance penalty:

1. `font.glyph_advance(ord("Ä")) * fontsize`
2. `font.text_length("Ä", fontsize=fontsize)`

For multi-character strings, the method offers a huge performance advantage compared to the previous implementation: instead of about 0.5 microseconds for each character, only 12.5 nanoseconds are required for the second and subsequent ones.

char_lengths(text, fontsize=11)

New in v1.18.14

Sequence of character lengths in points of a unicode string.

Parameters

- **text** (*str*) – a text string, UTF-8 encoded.
- **fontsize** (*float*) – the fontsize.

Return type

tuple

Returns

the lengths in points of the characters of a string when stored in the PDF. It works like `Font.text_length()` broken down to single characters. This is a high speed method, used e.g. in `TextWriter.fill_textbox()`. The following is true (allowing rounding errors): `font.text_length(text) == sum(font.char_lengths(text))`.

```
>>> font = fitz.Font("helv")
>>> text = "PyMuPDF"
>>> font.text_length(text)
50.115999937057495
>>> fitz.get_text_length(text, fontname="helv")
50.115999937057495
>>> sum(font.char_lengths(text))
50.115999937057495
>>> pprint(font.char_lengths(text))
(7.336999952793121, # P
5.5, # y
9.163000047206879, # M
6.115999937057495, # u
7.336999952793121, # P
7.942000031471252, # D
6.721000015735626) # F
```

buffer

- New in v1.17.6

Copy of the binary font file content.

Return type

bytes

flags

A dictionary with various font properties, each represented as bools. Example for Helvetica:

```
>>> pprint(font.flags)
{'bold': 0,
 'fake-bold': 0,
 'fake-italic': 0,
 'invalid-bbox': 0,
 'italic': 0,
 'mono': 0,
 'opentype': 0,
 'serif': 1,
 'stretch': 0,
 'substitute': 0}
```

Return type

dict

name

Return type

str

Name of the font. May be "" or "(null)".

bbox

The font bbox. This is the maximum of its glyph bboxes.

Return type

Rect

glyph_count

Return type

int

The number of glyphs defined in the font.

ascender

- New in v1.18.0

The ascender value of the font, see [here](#) for details. Please note that there is a difference to the strict definition: our value includes everything above the baseline – not just the height difference between upper case “A” and lower case “a”.

Return type

float

descender

- New in v1.18.0

The descender value of the font, see [here](#) for details. This value always is negative and is the portion that some glyphs descend below the base line, for example “g” or “y”. As a consequence, the value `ascender - descender` is the total height, that every glyph of the font fits into. This is true at least for most fonts – as always, there are exceptions, especially for calligraphic fonts, etc.

Return type

float

is_writable

- New in v1.18.0

Indicates whether this font can be used with *TextWriter*.

Return type

bool

6.8 Identity

Identity is a *Matrix* that performs no action – to be used whenever the syntax requires a matrix, but no actual transformation should take place. It has the form *fitz.Matrix(1, 0, 0, 1, 0, 0)*.

Identity is a constant, an “immutable” object. So, all of its matrix properties are read-only and its methods are disabled.

If you need a **mutable** identity matrix as a starting point, use one of the following statements:

```
>>> m = fitz.Matrix(1, 0, 0, 1, 0, 0) # specify the values
>>> m = fitz.Matrix(1, 1)             # use scaling by factor 1
>>> m = fitz.Matrix(0)                # use rotation by zero degrees
>>> m = fitz.Matrix(fitz.Identity)    # make a copy of Identity
```


6.9 IRect

IRect is a rectangular bounding box, very 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, e.g. considerations concerning emptiness and validity of rectangles also apply to this class. Methods and attributes have the same names, and in many cases are implemented by re-using the respective [Rect](#) counterparts.

Attribute / Method	Short Description
IRect.contains()	checks containment of another object
IRect.get_area()	calculate rectangle area
IRect.intersect()	common part with another rectangle
IRect.intersects()	checks for non-empty intersection
IRect.morph()	transform with a point and a matrix
IRect.torect()	matrix that transforms to another rectangle
IRect.norm()	the Euclidean norm
IRect.normalize()	makes a rectangle finite
IRect.bottom_left	bottom left point, synonym <i>bl</i>
IRect.bottom_right	bottom right point, synonym <i>br</i>
IRect.height	height of the rectangle
IRect.is_empty	whether rectangle is empty
IRect.is_infinite	whether rectangle is infinite
IRect.rect	the Rect equivalent
IRect.top_left	top left point, synonym <i>tl</i>
IRect.top_right	top right point, synonym <i>tr</i>
IRect.quad	Quad made from rectangle corners
IRect.width	width of the rectangle
IRect.x0	X-coordinate of the top left corner
IRect.x1	X-coordinate of the bottom right corner
IRect.y0	Y-coordinate of the top left corner
IRect.y1	Y-coordinate of the bottom right corner

Class API

class IRect

`__init__(self)`

`__init__(self, x0, y0, x1, y1)`

`__init__(self, irect)`

`__init__(self, sequence)`

Overloaded constructors. Also see examples below and those for the [Rect](#) class.

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

If sequence is specified, it must be a Python sequence type of 4 numbers (see [Using Python Sequences as Arguments in PyMuPDF](#)). Non-integer numbers will be truncated, non-numeric values will raise an exception.

The other parameters mean integer coordinates.

`get_area([unit])`

Calculates the area of the rectangle and, with no parameter, equals *abs(IRect)*. Like an empty rectangle, the area of an infinite rectangle is also zero.

Parameters

unit (*str*) – Specify required unit: respective squares of “px” (pixels, default), “in” (inches), “cm” (centimeters), or “mm” (millimeters).

Return type

float

intersect(*ir*)

The intersection (common rectangular area) of the current rectangle and *ir* is calculated and replaces the current rectangle. If either rectangle is empty, the result is also empty. If either rectangle is infinite, the other one is taken as the result – and hence also infinite if both rectangles were infinite.

Parameters

ir (*rect_like*) – Second rectangle.

contains(*x*)

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

Parameters

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

Return type

bool

intersects(*r*)

Checks whether the rectangle and the *rect_like* “*r*” contain a common non-empty *IRect*. This will always be *False* if either is infinite or empty.

Parameters

r (*rect_like*) – the rectangle to check.

Return type

bool

torect(*rect*)

- New in version 1.19.3

Compute the matrix which transforms this rectangle to a given one. See *Rect.torect()*.

Parameters

rect (*rect_like*) – the target rectangle. Must not be empty or infinite.

Return type

Matrix

Returns

a matrix *mat* such that *self* * *mat* = *rect*. Can for example be used to transform between the page and the pixmap coordinates.

morph(*fixpoint*, *matrix*)

- New in version 1.17.0

Return a new quad after applying a matrix to it using a fixed point.

Parameters

- **fixpoint** (*point_like*) – the fixed point.

- **matrix** (*matrix_like*) – the matrix.

Returns

a new *Quad*. This a wrapper of the same-named quad method. If infinite, the infinite quad is returned.

norm()

- New in version 1.16.0

Return the Euclidean norm of the rectangle treated as a vector of four numbers.

normalize()

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

top_left**tl**

Equals *Point*(*x0*, *y0*).

Type

Point

top_right**tr**

Equals *Point*(*x1*, *y0*).

Type

Point

bottom_left**bl**

Equals *Point*(*x0*, *y1*).

Type

Point

bottom_right**br**

Equals *Point*(*x1*, *y1*).

Type

Point

rect

The *Rect* with the same coordinates as floats.

Type

Rect

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

is_infinite

True if rectangle is infinite, *False* otherwise.

Type

bool

is_empty

True if rectangle is empty, *False* otherwise.

Type

bool

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
 - Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.
-

6.10 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.first_link` 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.set_border()</code>	modify border properties
<code>Link.set_colors()</code>	modify color properties
<code>Link.set_flags()</code>	modify link flags
<code>Link.border</code>	border characteristics
<code>Link.colors</code>	border line color
<code>Link.dest</code>	points to destination details
<code>Link.is_external</code>	external destination?
<code>Link.flags</code>	link annotation flags
<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>	<code>xref</code> number of the entry

Class API

class Link

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

PDF only: Change border width and dashing properties.

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

Parameters

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

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

PDF only: Changes the “stroke” color.

Note: In PDF, links are a subtype of annotations technically and **do not support fill colors**. However, to keep a consistent API, we do allow specifying a `fill=` parameter like with all annotations, which will be ignored with a warning.

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

Parameters

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

set_flags(flags)

New in v1.18.16

Set the PDF /F property of the link annotation. See [Annot.set_flags\(\)](#) for details. If not a PDF, this method is a no-op.

flags

New in v1.18.16

Return the link annotation flags, an integer (see [Annot.flags](#) for details). Zero if not a PDF.

colors

Meaningful for PDF only: A dictionary of two tuples 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. As mentioned above, the fill color is always *None* for links. The stroke color is used for the border of the link rectangle. The length of the tuple implicitly determines the colorspace: 1 = GRAY, 3 = RGB, 4 = CMYK. So (1.0, 0.0, 0.0) stands for RGB color red. The value of each float f is mapped to the integer value i in range 0 to 255 via the computation $f = i / 255$.

Return type

dict

border

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

- *width* – a float indicating the border thickness in points. The value is -1.0 if no width is specified.
- *dashes* – a sequence of integers specifying a line dash pattern. *[]* means no dashes, *[n]* means equal on-off lengths of n points, longer lists will be interpreted as specifying alternating on-off length values. See the [Adobe PDF References](#) page 126 for more detail.
- *style* – 1-byte border style: *S* (Solid) = solid rectangle surrounding the annotation, *D* (Dashed) = dashed rectangle surrounding the link, the dash pattern is specified by the *dashes* entry, *B* (Beveled) = a simulated embossed rectangle that appears to be raised above the surface of the page, *I* (Inset) = a simulated engraved rectangle that appears to be recessed below the surface of the page, *U* (Underline) = a single line along the bottom of the annotation rectangle.

Return type

dict

rect

The area that can be clicked in untransformed coordinates.

Type

[Rect](#)

isExternal

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

Type

bool

uri

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

Type

str

xref

An integer specifying the PDF *xref*. Zero if not a PDF.

Type

int

next

The next link or *None*.

Type*Link***dest**

The link destination details object.

Type*linkDest*

6.11 linkDest

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

Note: Up to MuPDF v1.9.0 this class existed inside MuPDF and was dropped in version 1.10.0. For backward compatibility, PyMuPDF is still maintaining it, although some of its attributes are no longer backed by data actually available via MuPDF.

Attribute	Short Description
<i>linkDest.dest</i>	destination
<i>linkDest.fileSpec</i>	file specification (path, filename)
<i>linkDest.flags</i>	descriptive flags
<i>linkDest.isMap</i>	is this a MAP?
<i>linkDest.isUri</i>	is this a URI?
<i>linkDest.kind</i>	kind of destination
<i>linkDest.lt</i>	top left coordinates
<i>linkDest.named</i>	name if named destination
<i>linkDest.newWindow</i>	name of new window
<i>linkDest.page</i>	page number
<i>linkDest.rb</i>	bottom right coordinates
<i>linkDest.uri</i>	URI

Class API**class linkDest****dest**

Target destination name if *linkDest.kind* is *LINK_GOTOR* and *linkDest.page* is *-1*.

Type

str

fileSpec

Contains the filename and path this link points to, if *linkDest.kind* is *LINK_GOTOR* or *LINK_LAUNCH*.

Type

str

flags

A bitfield describing the validity and meaning of the different aspects of the destination. As far as possible, link destinations are constructed such that e.g. *linkDest.lt* and *linkDest.rb* can be treated as defining a bounding box. But the flags indicate which of the values were actually specified, see *Link Destination Flags*.

Type

int

isMap

This flag specifies whether to track the mouse position when the URI is resolved. Default value: False.

Type

bool

isUri

Specifies whether this destination is an internet resource (as opposed to e.g. a local file specification in URI format).

Type

bool

kind

Indicates the type of this destination, like a place in this document, a URI, a file launch, an action or a place in another file. Look at *Link Destination Kinds* to see the names and numerical values.

Type

int

lt

The top left *Point* of the destination.

Type

Point

named

This destination refers to some named action to perform (e.g. a javascript, see *Adobe PDF References*). Standard actions provided are *NextPage*, *PrevPage*, *FirstPage*, and *LastPage*.

Type

str

newWindow

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

Type
bool

page

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

Type
int

rb

The bottom right *Point* of this destination.

Type
Point

uri

The name of the URI this destination points to.

Type
str

6.12 Matrix

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

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

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

Please note:

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

Method / Attribute	Description
<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.norm()</i>	the Euclidean norm
<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.is_rectilinear</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*, *l*)

__init__(*self*, *a*, *b*, *c*, *d*, *e*, *f*)

__init__(*self*, *matrix*)

__init__(*self*, *degree*)

__init__(*self*, *sequence*)

Overloaded constructors.

Without parameters, the zero matrix *Matrix(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)* will be created.

*zoom-** and *shear-** specify zoom or shear values (float) and create a zoom or shear matrix, respectively.

For “matrix” a **new copy** of another matrix will be made.

Float value “degree” specifies the creation of a rotation matrix which rotates anit-clockwise.

A “sequence” must be any Python sequence object with exactly 6 float entries (see [Using Python Sequences as Arguments in PyMuPDF](#)).

fitz.Matrix(1, 1), *fitz.Matrix(0.0* and **fitz.Matrix(fitz.Identity)* create modifyable versions of the *Identity* matrix, which looks like *[1, 0, 0, 1, 0, 0]*.

norm()

- New in version 1.16.0

Return the Euclidean norm of the matrix as a vector.

prerotate(*deg*)

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

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

Parameters

deg (*float*) – The rotation angle in degrees (use conventional notation based on Pi = 180 degrees).

prescale(*sx*, *sy*)

Modify the matrix to scale by the zoom factors *sx* and *sy*. Has effects on attributes *a* thru *d* only: $[a, b, c, d, e, f] \rightarrow [a*sx, b*sx, c*sy, d*sy, e, f]$.

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: $[a, b, c, d, e, f] \rightarrow [c*sy, d*sy, a*sx, b*sx, e, f]$.

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: $[a, b, c, d, e, f] \rightarrow [a, b, c, d, tx*a + ty*c, tx*b + ty*d]$.

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

is_rectilinear

Rectilinear means that no shearing is present and that any rotations are integer multiples of 90 degrees. Usually this is used to confirm that (axis-aligned) rectangles before the transformation are still axis-aligned rectangles afterwards.

Type
bool

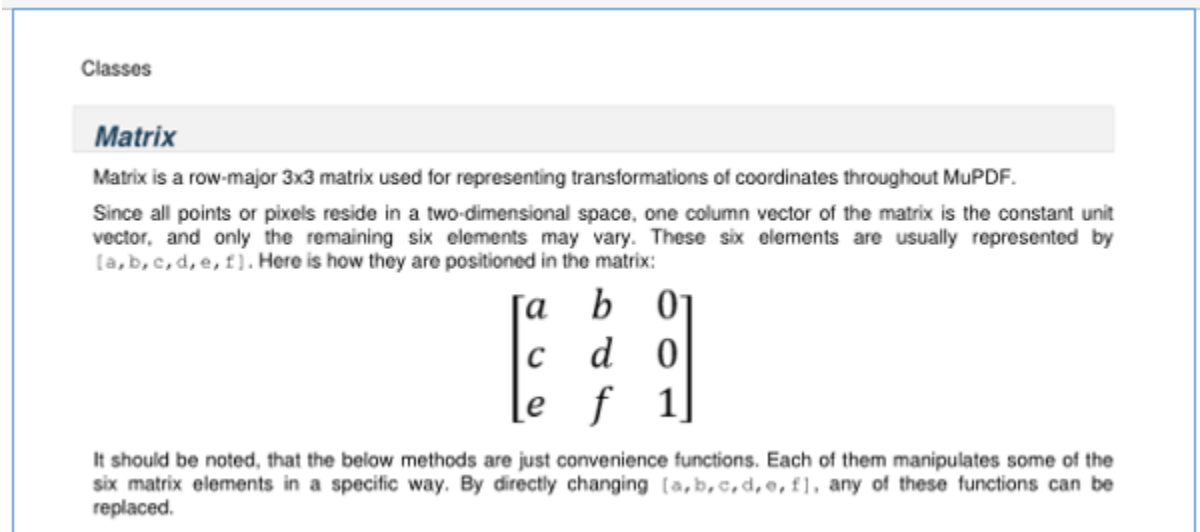
Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
 - A matrix can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.
 - Changes of matrix properties and execution of matrix methods can be executed consecutively. This is the same as multiplying the respective matrices.
 - Matrix multiplication is **not commutative** – changing the execution sequence in general changes the result. So it can quickly become unclear which result a transformation will yield.
-

6.12.1 Examples

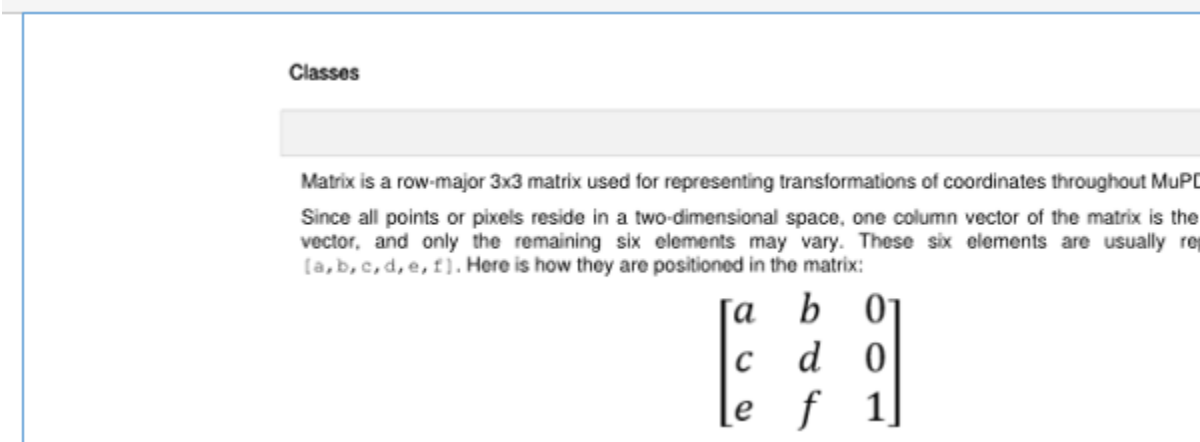
Here are examples to illustrate some of the effects achievable. The following pictures start with a page of the PDF version of this help file. We show what happens when a matrix is being applied (though always full pages are created, only parts are displayed here to save space).

This is the original page image:



6.12.2 Shifting

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



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

Classes

Matrix

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

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

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

6.12.3 Flipping

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

Classes

Matrix

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

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

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

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

$$\begin{bmatrix} e & t & 1 \\ c & q & 0 \\ a & p & 0 \end{bmatrix}$$

[e t c q a p]. 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 q since all points or pixels reside in a two-dimensional space, one column vector of the matrix is the constant unit matrix is a row-major 3x3 matrix used for representing transformations of coordinates throughout MuPDF.

Matrix

C192262

6.12.4 Shearing

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



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

Classes

Matrix

Matrix is a row-major 3x3 matrix used for image transformations in MuPDF. With matrices you can manipulate the rendered image of a page in a variety of ways: (parts of) pages can be rotated, zoomed, flipped, sheared and shifted by setting some or all of just six numerical values.

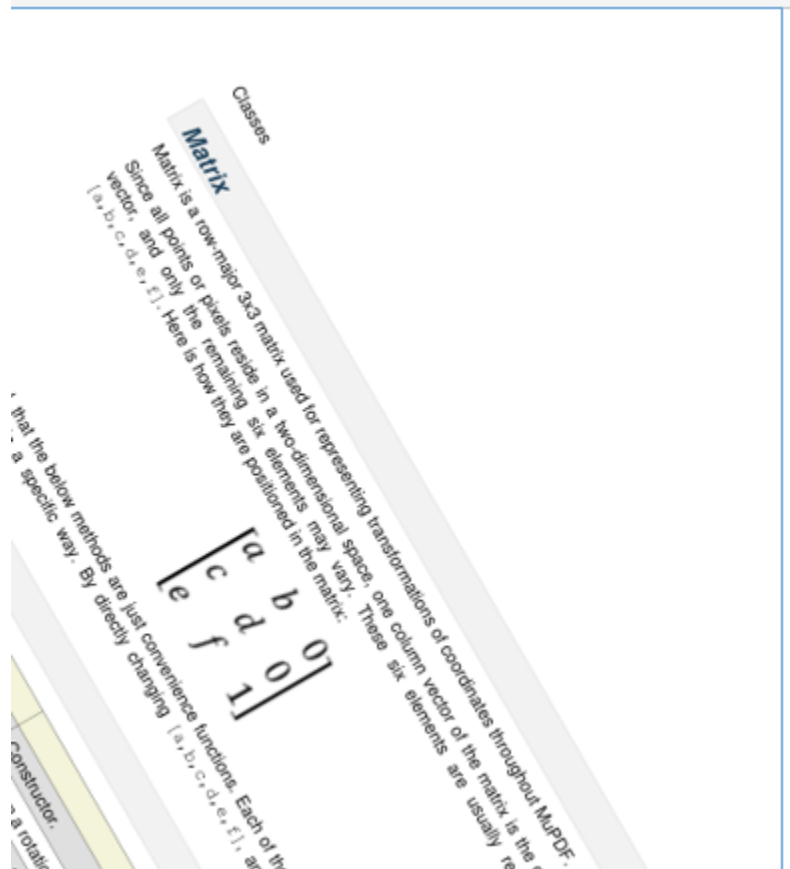
Since all points or pixels live in a two-dimensional space, one column vector of that matrix is a constant unit vector, and only the remaining six elements are used for manipulations. These six elements are usually represented by $[a, b, c, d, e, f]$. Here is how they are positioned in the matrix:

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

It should be noted, that

6.12.5 Rotating

Finally a rotation by 30 clockwise degrees (*prerotate(-30)*).



6.13 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 outline target
<i>Outline.is_external</i>	target outside document
<i>Outline.is_open</i>	whether sub-outlines are open or collapsed
<i>Outline.dest</i>	points to destination details object

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

Indicator showing whether any sub-outlines should be expanded (*True*) or be collapsed (*False*). This information is interpreted by PDF reader software.

Type

bool

is_external

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

Type

bool

uri

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

Type*str***dest**

The link destination details object.

Type*linkDest*

6.14 Page

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

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

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

6.14.1 Modifying Pages

Changing page properties and adding or changing page content is available for PDF documents only.

In a nutshell, this is what you can do with PyMuPDF:

- Modify page rotation and the visible part (“cropbox”) of the page.
- Insert images, other PDF pages, text and simple geometrical objects.
- Add annotations and form fields.

Note: Methods require coordinates (points, rectangles) to put content in desired places. Please be aware that since v1.17.0 these coordinates **must always** be provided relative to the **unrotated** page. The reverse is also true: except *Page.rect*, resp. *Page.bound()* (both *reflect* when the page is rotated), all coordinates returned by methods and attributes pertain to the unrotated page.

So the returned value of e.g. *Page.get_image_bbox()* will not change if you do a *Page.set_rotation()*. The same is true for coordinates returned by *Page.get_text()*, annotation rectangles, and so on. If you want to find out, where an object is located in **rotated coordinates**, multiply the coordinates with *Page.rotation_matrix*. There also is its inverse, *Page.derotation_matrix*, which you can use when interfacing with other readers, which may behave differently in this respect.

Note: If you add or update annotations, links or form fields on the page and immediately afterwards need to work with them (i.e. **without leaving the page**), you should reload the page using `Document.reload_page()` before referring to these new or updated items.

Reloading the page is generally recommended – although not strictly required in all cases. However, some annotation and widget types have extended features in PyMuPDF compared to MuPDF. More of these extensions may also be added in the future.

Reloading the page ensures all your changes have been fully applied to PDF structures, so you can safely create Pixmaps or successfully iterate over annotations, links and form fields.

Method / Attribute	Short Description
<code>Page.add_caret_annot()</code>	PDF only: add a caret annotation
<code>Page.add_circle_annot()</code>	PDF only: add a circle annotation
<code>Page.add_file_annot()</code>	PDF only: add a file attachment annotation
<code>Page.add_freetext_annot()</code>	PDF only: add a text annotation
<code>Page.add_highlight_annot()</code>	PDF only: add a “highlight” annotation
<code>Page.add_ink_annot()</code>	PDF only: add an ink annotation
<code>Page.add_line_annot()</code>	PDF only: add a line annotation
<code>Page.add_polygon_annot()</code>	PDF only: add a polygon annotation
<code>Page.add_polyline_annot()</code>	PDF only: add a multi-line annotation
<code>Page.add_rect_annot()</code>	PDF only: add a rectangle annotation
<code>Page.add_redact_annot()</code>	PDF only: add a redaction annotation
<code>Page.add_squiggly_annot()</code>	PDF only: add a “squiggly” annotation
<code>Page.add_stamp_annot()</code>	PDF only: add a “rubber stamp” annotation
<code>Page.add_strikeout_annot()</code>	PDF only: add a “strike-out” annotation
<code>Page.add_text_annot()</code>	PDF only: add a comment
<code>Page.add_underline_annot()</code>	PDF only: add an “underline” annotation
<code>Page.add_widget()</code>	PDF only: add a PDF Form field
<code>Page.annot_names()</code>	PDF only: a list of annotation (and widget) names
<code>Page.annot_xrefs()</code>	PDF only: a list of annotation (and widget) xrefs
<code>Page.annots()</code>	return a generator over the annots on the page
<code>Page.apply_redactions()</code>	PDF only: process the redactions of the page
<code>Page.bound()</code>	rectangle of the page
<code>Page.delete_annot()</code>	PDF only: delete an annotation
<code>Page.delete_image()</code>	PDF only: delete an image
<code>Page.delete_link()</code>	PDF only: delete a link
<code>Page.delete_widget()</code>	PDF only: delete a widget / field
<code>Page.draw_bezier()</code>	PDF only: draw a cubic Bezier curve
<code>Page.draw_circle()</code>	PDF only: draw a circle
<code>Page.draw_curve()</code>	PDF only: draw a special Bezier curve
<code>Page.draw_line()</code>	PDF only: draw a line
<code>Page.draw_oval()</code>	PDF only: draw an oval / ellipse
<code>Page.draw_polyline()</code>	PDF only: connect a point sequence
<code>Page.draw_quad()</code>	PDF only: draw a quad
<code>Page.draw_rect()</code>	PDF only: draw a rectangle
<code>Page.draw_sector()</code>	PDF only: draw a circular sector
<code>Page.draw_squiggle()</code>	PDF only: draw a squiggly line
<code>Page.draw_zigzag()</code>	PDF only: draw a zig-zagged line
<code>Page.get_drawings()</code>	get list of the draw commands contained in the page

continues on next page

Table 3 – continued from previous page

Method / Attribute	Short Description
<code>Page.get_fonts()</code>	PDF only: get list of referenced fonts
<code>Page.get_image_bbox()</code>	PDF only: get bbox and matrix of embedded image
<code>Page.get_image_info()</code>	get list of meta information for all used images
<code>Page.get_image_rects()</code>	PDF only: improved version of <code>Page.get_image_bbox()</code>
<code>Page.get_images()</code>	PDF only: get list of referenced images
<code>Page.get_label()</code>	PDF only: return the label of the page
<code>Page.get_links()</code>	get all links
<code>Page.get_pixmap()</code>	create a page image in raster format
<code>Page.get_svg_image()</code>	create a page image in SVG format
<code>Page.get_text()</code>	extract the page's text
<code>Page.get_textbox()</code>	extract text contained in a rectangle
<code>Page.get_textpage_ocr()</code>	create a <code>TextPage</code> with OCR for the page
<code>Page.get_textpage()</code>	create a <code>TextPage</code> for the page
<code>Page.get_xobjects()</code>	PDF only: get list of referenced xobjects
<code>Page.insert_font()</code>	PDF only: insert a font for use by the page
<code>Page.insert_image()</code>	PDF only: insert an image
<code>Page.insert_link()</code>	PDF only: insert a link
<code>Page.insert_text()</code>	PDF only: insert text
<code>Page.insert_textbox()</code>	PDF only: insert a text box
<code>Page.links()</code>	return a generator of the links on the page
<code>Page.load_annot()</code>	PDF only: load a specific annotation
<code>Page.load_widget()</code>	PDF only: load a specific field
<code>Page.load_links()</code>	return the first link on a page
<code>Page.new_shape()</code>	PDF only: create a new <i>Shape</i>
<code>Page.replace_image()</code>	PDF only: replace an image
<code>Page.search_for()</code>	search for a string
<code>Page.set_artbox()</code>	PDF only: modify <code>/ArtBox</code>
<code>Page.set_bleedbox()</code>	PDF only: modify <code>/BleedBox</code>
<code>Page.set_cropbox()</code>	PDF only: modify the cropbox (visible page)
<code>Page.set_mediabox()</code>	PDF only: modify <code>/MediaBox</code>
<code>Page.set_rotation()</code>	PDF only: set page rotation
<code>Page.set_trimbox()</code>	PDF only: modify <code>/TrimBox</code>
<code>Page.show_pdf_page()</code>	PDF only: display PDF page image
<code>Page.update_link()</code>	PDF only: modify a link
<code>Page.widgets()</code>	return a generator over the fields on the page
<code>Page.write_text()</code>	write one or more <i>TextWriter</i> objects
<code>Page.cropbox_position</code>	displacement of the cropbox
<code>Page.cropbox</code>	the page's cropbox
<code>Page.artbox</code>	the page's <code>/ArtBox</code>
<code>Page.bleedbox</code>	the page's <code>/BleedBox</code>
<code>Page.trimbox</code>	the page's <code>/TrimBox</code>
<code>Page.derotation_matrix</code>	PDF only: get coordinates in unrotated page space
<code>Page.first_annot</code>	first <i>Annot</i> on the page
<code>Page.first_link</code>	first <i>Link</i> on the page
<code>Page.first_widget</code>	first widget (form field) on the page
<code>Page.mediabox_size</code>	bottom-right point of <code>mediabox</code>
<code>Page.mediabox</code>	the page's <code>mediabox</code>
<code>Page.number</code>	page number
<code>Page.parent</code>	owning document object

continues on next page

Table 3 – continued from previous page

Method / Attribute	Short Description
<code>Page.rect</code>	rectangle of the page
<code>Page.rotation_matrix</code>	PDF only: get coordinates in rotated page space
<code>Page.rotation</code>	PDF only: page rotation
<code>Page.transformation_matrix</code>	PDF only: translate between PDF and MuPDF space
<code>Page.xref</code>	PDF only: page <i>xref</i>

Class API**class Page****bound()**

Determine the rectangle of the page. Same as property `Page.rect` below. For PDF documents this **usually** also coincides with `mediabox` and `cropbox`, but not always. For example, if the page is rotated, then this is reflected by this method – the `Page.cropbox` however will not change.

Return type*Rect***add_caret_annot(*point*)**

- New in v1.16.0

PDF only: Add a caret icon. A caret annotation is a visual symbol normally used to indicate the presence of text edits on the page.

Parameters

point (*point_like*) – the top left point of a 20 x 20 rectangle containing the MuPDF-provided icon.

Return type*Annot***Returns**

the created annotation. Stroke color blue = (0, 0, 1), no fill color support.

**'Caret' annotation****add_text_annot(*point*, *text*, *icon*='Note')**

PDF only: Add a comment icon (“sticky note”) with accompanying text. Only the icon is visible, the accompanying text is hidden and can be visualized by many PDF viewers by hovering the mouse over the symbol.

Parameters

- **point** (*point_like*) – the top left point of a 20 x 20 rectangle containing the MuPDF-provided “note” icon.
- **text** (*str*) – the commentary text. This will be shown on double clicking or hovering over the icon. May contain any Latin characters.
- **icon** (*str*) – (*new in v1.16.0*) choose one of “Note” (default), “Comment”, “Help”, “Insert”, “Key”, “NewParagraph”, “Paragraph” as the visual symbol for the embodied text⁴.

⁴ You are generally free to choose any of the *Annotation Icons in MuPDF* you consider adequate.

Return type*Annot***Returns**

the created annotation. Stroke color yellow = (1, 1, 0), no fill color support.

```
add_freetext_annot(rect, text, fontsize=12, fontname='helv', border_color=None, text_color=0,
                    fill_color=1, rotate=0, align=TEXT_ALIGN_LEFT)
```

- Changed in v1.19.6: add border color parameter

PDF only: Add text in a given rectangle.

Parameters

- **rect** (*rect_like*) – the rectangle into which the text should be inserted. Text is automatically wrapped to a new line at box width. Lines not fitting into the box will be invisible.
- **text** (*str*) – the text. (*New in v1.17.0*) May contain any mixture of Latin, Greek, Cyrillic, Chinese, Japanese and Korean characters. The respective required font is automatically determined.
- **fontsize** (*float*) – the font size. Default is 12.
- **fontname** (*str*) – the font name. Default is “Helv”. Accepted alternatives are “Cour”, “TiRo”, “ZaDb” and “Symb”. The name may be abbreviated to the first two characters, like “Co” for “Cour”. Lower case is also accepted. (*Changed in v1.16.0*) Bold or italic variants of the fonts are **no longer accepted**. A user-contributed script provides a circumvention for this restriction – see section *Using Buttons and JavaScript* in chapter FAQ. (*New in v1.17.0*) The actual font to use is now determined on a by-character level, and all required fonts (or sub-fonts) are automatically included. Therefore, you should rarely ever need to care about this parameter and let it default (except you insist on a serified font for your non-CJK text parts).
- **text_color** (*sequence, float*) – (*new in v1.16.0*) the text color. Default is black.
- **fill_color** (*sequence, float*) – (*new in v1.16.0*) the fill color. Default is white.
- **text_color** – the text color. Default is black.
- **border_color** (*sequence, float*) – (*new in v1.19.6*) the border color. Default is None.
- **align** (*int*) – (*new in v1.17.0*) text alignment, one of TEXT_ALIGN_LEFT, TEXT_ALIGN_CENTER, TEXT_ALIGN_RIGHT - justify is **not supported**.
- **rotate** (*int*) – the text orientation. Accepted values are 0, 90, 270, invalid entries are set to zero.

Return type*Annot***Returns**

the created annotation. Color properties **can only be changed** using special parameters of *Annot.update()*. There, you can also set a border color different from the text color.

```
add_file_annot(pos, buffer, filename, ufilename=None, desc=None, icon='PushPin')
```

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

Parameters

- **pos** (*point_like*) – the top-left point of a 18x18 rectangle containing the MuPDF-provided “PushPin” icon.
- **buffer** (*bytes, bytearray, BytesIO*) – the data to be stored (actual file content, any data, etc.).

Changed in v1.14.13 *io.BytesIO* is now also supported.

- **filename** (*str*) – the filename to associate with the data.
- **ufilename** (*str*) – the optional PDF unicode version of filename. Defaults to filename.
- **desc** (*str*) – an optional description of the file. Defaults to filename.
- **icon** (*str*) – (*new in v1.16.0*) choose one of “PushPin” (default), “Graph”, “Paperclip”, “Tag” as the visual symbol for the attached data^{Page 229, 4}.

Return type

Annot

Returns

the created annotation. Stroke color yellow = (1, 1, 0), no fill color support.

add_ink_annot(*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 =(0, 0, 0), line width 1. No fill color support.

add_line_annot(*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 (stroke) color red = (1, 0, 0) and line width 1. No fill color support. The **annot rectangle** is automatically created to contain both points, each one surrounded by a circle of radius 3 * line width to make room for any line end symbols.

add_rect_annot(*rect*)

add_circle_annot(*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 (stroke) color red = (1, 0, 0), line width 1, fill color is supported.

add_redact_annot (*quad*, *text=None*, *fontname=None*, *fontsize=11*, *align=TEXT_ALIGN_LEFT*, *fill=(1, 1, 1)*, *text_color=(0, 0, 0)*, *cross_out=True*)

- New in v1.16.11

PDF only: Add a redaction annotation. A redaction annotation identifies content to be removed from the document. Adding such an annotation is the first of two steps. It makes visible what will be removed in the subsequent step, [Page.apply_redactions\(\)](#).

Parameters

- **quad** (*quad_like*, *rect_like*) – specifies the (rectangular) area to be removed which is always equal to the annotation rectangle. This may be a [rect_like](#) or [quad_like](#) object. If a quad is specified, then the envelopping rectangle is taken.
- **text** (*str*) – (New in v1.16.12) text to be placed in the rectangle after applying the redaction (and thus removing old content).
- **fontname** (*str*) – (New in v1.16.12) the font to use when *text* is given, otherwise ignored. The same rules apply as for [Page.insert_textbox\(\)](#) – which is the method [Page.apply_redactions\(\)](#) internally invokes. The replacement text will be **vertically centered**, if this is one of the CJK or *PDF Base 14 Fonts*.

Note:

- For an **existing** font of the page, use its reference name as *fontname* (this is *item[4]* of its entry in [Page.get_fonts\(\)](#)).
- For a **new, non-builtin** font, proceed as follows:

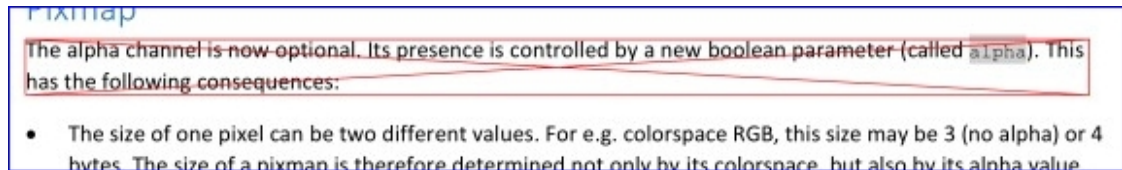
```
page.insert_text(point, # anywhere, but outside all_
↳redaction rectangles
    "something", # some non-empty string
    fontname="newname", # new, unused reference name
    fontfile="...", # desired font file
    render_mode=3, # makes the text invisible
)
page.add_redact_annot(..., fontname="newname")
```

- **fontsize** (*float*) – (New in v1.16.12) the fontsize to use for the replacing text. If the text is too large to fit, several insertion attempts will be made, gradually reducing the fontsize to no less than 4. If then the text will still not fit, no text insertion will take place at all.
- **align** (*int*) – (New in v1.16.12) the horizontal alignment for the replacing text. See [insert_textbox\(\)](#) for available values. The vertical alignment is (approximately) centered if a PDF built-in font is used (CJK or *PDF Base 14 Fonts*).

- **fill** (*sequence*) – (New in v1.16.12) the fill color of the rectangle **after applying** the redaction. The default is *white* = (1, 1, 1), which is also taken if *None* is specified. (Changed in v1.16.13) To suppress a fill color altogether, specify *False*. In this cases the rectangle remains transparent.
- **text_color** (*sequence*) – (New in v1.16.12) the color of the replacing text. Default is *black* = (0, 0, 0).
- **cross_out** (*bool*) – (new in v1.17.2) add two diagonal lines to the annotation rectangle.

Return type*Annot***Returns**

the created annotation. (Changed in v1.17.2) Its standard appearance looks like a red rectangle (no fill color), optionally showing two diagonal lines. Colors, line width, dashing, opacity and blend mode can now be set and applied via *Annot.update()* like with other annotations.

**add_polyline_annot**(*points*)**add_polygon_annot**(*points*)

PDF only: Add an annotation consisting of lines which connect the given points. A **Polygon**'s first and last points are automatically connected, which does not happen for a **PolyLine**. The **rectangle** is automatically created as the smallest rectangle containing the points, each one surrounded by a circle of radius 3 (= 3 * line width). The following shows a 'PolyLine' that has been modified with colors and line ends.

Parameters

points (*list*) – a list of *point_like* objects.

Return type*Annot***Returns**

the created annotation. It is drawn with line color black, line width 1 no fill color but fill color support. Use methods of *Annot* to make any changes to achieve something like this:

**PolyLine annotation****add_underline_annot**(*quads=None, start=None, stop=None, clip=None*)**add_strikeout_annot**(*quads=None, start=None, stop=None, clip=None*)**add_squiggly_annot**(*quads=None, start=None, stop=None, clip=None*)**add_highlight_annot**(*quads=None, start=None, stop=None, clip=None*)

PDF only: These annotations are normally used for **marking text** which has previously been somehow located (for example via *Page.search_for()*). But this is not required: you are free to “mark” just anything.

Standard (stroke only – no fill color support) colors are chosen per annotation type: **yellow** for highlighting, **red** for striking out, **green** for underlining, and **magenta** for wavy underlining.

All these four methods convert the arguments into a list of *Quad* objects. The **annotation** rectangle is then calculated to envelop all these quadrilaterals.

Note: *search_for()* delivers a list of either *Rect* or *Quad* objects. Such a list can be directly used as an argument for these annotation types and will deliver **one common annotation** for all occurrences of the search string:

```
>>> # prefer quads=True in text searching for annotations!
>>> quads = page.search_for("pymupdf", quads=True)
>>> page.add_highlight_annot(quads)
```

Note: Obviously, text marker annotations need to know what is the top, the bottom, the left, and the right side of the area(s) to be marked. If the arguments are quads, this information is given by the sequence of the quad points. In contrast, a rectangle delivers much less information – this is illustrated by the fact, that $4! = 24$ different quads can be constructed with the four corners of a rectangle.

Therefore, we **strongly recommend** to use the quads option for text searches, to ensure correct annotations. A similar consideration applies to marking **text spans** extracted with the “dict” / “rawdict” options of *Page.get_text()*. For more details on how to compute quadrilaterals in this case, see section “How to Mark Non-horizontal Text” of FAQ.

Parameters

- **quads** (*rect_like*, *quad_like*, *list*, *tuple*) – (Changed in v1.14.20) the location(s) – rectangle(s) or quad(s) – to be marked. A list or tuple must consist of *rect_like* or *quad_like* items (or even a mixture of either). Every item must be finite, convex and not empty (as applicable). (Changed in v1.16.14) **Set this parameter to None** if you want to use the following arguments.
- **start** (*point_like*) – (New in v1.16.14) start text marking at this point. Defaults to the top-left point of *clip*.
- **stop** (*point_like*) – (New in v1.16.14) stop text marking at this point. Defaults to the bottom-right point of *clip*.
- **clip** (*rect_like*) – (New in v1.16.14) only consider text lines intersecting this area. Defaults to the page rectangle.

Return type

Annot or (changed in v1.16.14) *None*

Returns

the created annotation. (Changed in v1.16.14) If *quads* is an empty list, **no annotation** is created.

Note: Starting with v1.16.14 you can use parameters *start*, *stop* and *clip* to highlight consecutive lines between the points *start* and *stop*. Make use of *clip* to further reduce the selected line bboxes and thus deal with e.g. multi-column pages. The following multi-line highlight on a page with three text columns was created by specifying the two red points and setting clip accordingly.



add_stamp_annot(rect, stamp=0)

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

Parameters

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

Note:

- The stamp’s text and its border line will automatically be sized and be put horizontally and vertically centered in the given rectangle. [Annot.rect](#) is automatically calculated to fit the given **width** and will usually be smaller than this parameter.
- The font chosen is “Times Bold” and the text will be upper case.
- The appearance can be changed using [Annot.set_opacity\(\)](#) and by setting the “stroke” color (no “fill” color supported).
- This can be used to create watermark images: on a temporary PDF page create a stamp annotation with a low opacity value, make a pixmap from it with *alpha=True* (and potentially also rotate it), discard the temporary PDF page and use the pixmap with [insert_image\(\)](#) for your target PDF.



'Stamp' annotation

add_widget(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.

delete_annot(annot)

- Changed in v1.16.6: The removal will now include any bound ‘Popup’ or response annotations and related objects.

PDF only: Delete 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. Please remember that physical removal requires saving to a new file with garbage > 0.

delete_widget(*widget*)

- New in v1.18.4

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

Parameters

widget (*Widget*) – the widget to be deleted.

Return type

Widget

Returns

the widget following the deleted one. Please remember that physical removal requires saving to a new file with garbage > 0.

apply_redactions(*images=PDF_REDACT_IMAGE_PIXELS*)

- New in v1.16.11
- Changed in v1.16.12: The previous *mark* parameter is gone. Instead, the respective rectangles are filled with the individual *fill* color of each redaction annotation. If a *text* was given in the annotation, then *insert_textbox()* is invoked to insert it, using parameters provided with the redaction.
- Changed in v1.18.0: added option for handling images that overlap redaction areas.

PDF only: Remove all **text content** contained in any redaction rectangle.

This method applies and then deletes all redactions from the page.

Parameters

images (*int*) – How to redact overlapping images. The default (2) blanks out overlapping pixels. *PDF_REDACT_IMAGE_NONE* (0) ignores, and *PDF_REDACT_IMAGE_REMOVE* (1) completely removes all overlapping images.

Returns

True if at least one redaction annotation has been processed, *False* otherwise.

Note:

- Text contained in a redaction rectangle will be **physically** removed from the page (assuming *Document.save()* with a suitable garbage option) and will no longer appear in e.g. text extractions or anywhere else. All redaction annotations will also be removed. Other annotations are unaffected.
- All overlapping links will be removed. If the rectangle of the link was covering text, then only the overlapping part of the text is being removed. Similar applies to images covered by link rectangles.
- (*Changed in v1.18.0*) The overlapping parts of **images** will be blanked-out for default option *PDF_REDACT_IMAGE_PIXELS*. Option 0 does not touch any images and 1 will remove any image with an overlap. Please be aware that there is a bug for option *PDF_REDACT_IMAGE_PIXELS* = 2: transparent images will be incorrectly handled!

- For option `images=PDF_REDACT_IMAGE_REMOVE` only this page's **references to the images** are removed - not necessarily the images themselves. Images are completely removed from the file only, if no longer referenced at all (assuming suitable garbage collection options).
- For option `images=PDF_REDACT_IMAGE_PIXELS` a new image of format PNG is created, which the page will use in place of the original one. The original image is not deleted or replaced as part of this process, so other pages may still show the original. In addition, the new, modified PNG image currently is **stored uncompressed**. Do keep these aspects in mind when choosing the right garbage collection method and compression options during save.
- **Text removal** is done by character: A character is removed if its bbox has a **non-empty overlap** with a redaction rectangle (*changed in MuPDF v1.17*). Depending on the font properties and / or the chosen line height, deletion may occur for undesired text parts. Using `Tools.set_small_glyph_heights()` with a `True` argument before text search may help to prevent this.
- Redactions are a simple way to replace single words in a PDF, or to just physically remove them. Locate the word "secret" using some text extraction or search method and insert a redaction using "xxxxxx" as replacement text for each occurrence.
 - Be wary if the replacement is longer than the original – this may lead to an awkward appearance, line breaks or no new text at all.
 - For a number of reasons, the new text may not exactly be positioned on the same line like the old one – especially true if the replacement font was not one of CJK or *PDF Base 14 Fonts*.

`delete_link(linkdict)`

PDF only: Delete the specified link from the page. The parameter must be an **original item** of `get_links()` (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.

`insert_link(linkdict)`

PDF only: Insert a new link on this page. The parameter must be a dictionary of format as provided by `get_links()` (see below).

Parameters

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

`update_link(linkdict)`

PDF only: Modify the specified link. The parameter must be a (modified) **original item** of `get_links()` (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.

Warning: If updating / inserting a URI link ("*kind*": `LINK_URI`), please make sure to start the value for the "*uri*" key with a disambiguating string like "`http://`", "`https://`", "`file://`", "`ftp://`", "`mailto:`", etc. Otherwise – depending on your browser or other "consumer" software – unexpected default assumptions may lead to unwanted behaviours.

`get_label()`

- New in v1.18.6

PDF only: Return the label for the page.

Return type

str

Returns

the label string like “vii” for Roman numbering or “” if not defined.

get_links()

Retrieves **all** links of a page.

Return type

list

Returns

A list of dictionaries. For a description of the dictionary entries see below. Always use this or the [Page.links\(\)](#) method if you intend to make changes to the links of a page.

links(kinds=None)

- New in v1.16.4

Return a generator over the page’s links. The results equal the entries of [Page.get_links\(\)](#).

Parameters

kinds (*sequence*) – a sequence of integers to down-select to one or more link kinds. Default is all links. Example: *kinds=(fitz.LINK_GOTO,)* will only return internal links.

Return type

generator

Returns

an entry of [Page.get_links\(\)](#) for each iteration.

annots(types=None)

- New in v1.16.4

Return a generator over the page’s annotations.

Parameters

types (*sequence*) – a sequence of integers to down-select to one or more annotation types. Default is all annotations. Example: *types=(fitz.PDF_ANNOT_FREETEXT, fitz.PDF_ANNOT_TEXT)* will only return ‘FreeText’ and ‘Text’ annotations.

Return type

generator

Returns

an [Annot](#) for each iteration.

Caution: You **cannot safely update annotations** from within this generator. This is because most annotation updates require reloading the page via `page = doc.reload_page(page)`. To circumvent this restriction, make a list of annotations xref numbers first and then iterate over these numbers:

```
In [4]: xrefs = [annot.xref for annot in page.annots(types=[...
→])]
In [5]: for xref in xrefs:
...:     annot = page.load_annot(xref)
...:     annot.update()
```

```
...:     page = doc.reload_page(page)
In [6]:
```

widgets(*types=None*)

- New in v1.16.4

Return a generator over the page's form fields.

Parameters

types (*sequence*) – a sequence of integers to down-select to one or more widget types. Default is all form fields. Example: `types=(fitz.PDF_WIDGET_TYPE_TEXT,)` will only return 'Text' fields.

Return type

generator

Returns

a *Widget* for each iteration.

write_text(*rect=None, writers=None, overlay=True, color=None, opacity=None, keep_proportion=True, rotate=0, oc=0*)

- New in v1.16.18

PDF only: Write the text of one or more *TextWriter* objects to the page.

Parameters

- **rect** (*rect_like*) – where to place the text. If omitted, the rectangle union of the text writers is used.
- **writers** (*sequence*) – a non-empty tuple / list of *TextWriter* objects or a single *TextWriter*.
- **opacity** (*float*) – set transparency, overwrites resp. value in the text writers.
- **color** (*sequ*) – set the text color, overwrites resp. value in the text writers.
- **overlay** (*bool*) – put the text in foreground or background.
- **keep_proportion** (*bool*) – maintain the aspect ratio.
- **rotate** (*float*) – rotate the text by an arbitrary angle.
- **oc** (*int*) – (new in v1.18.4) the *xref* of an *OCG* or *OCMD*.

Note: Parameters *overlay*, *keep_proportion*, *rotate* and *oc* have the same meaning as in *Page.show_pdf_page()*.

insert_text(*point, text, fontsize=11, fontname='helv', fontfile=None, idx=0, color=None, fill=None, render_mode=0, border_width=1, encoding=TEXT_ENCODING_LATIN, rotate=0, morph=None, stroke_opacity=1, fill_opacity=1, overlay=True, oc=0*)

- Changed in v1.18.4

PDF only: Insert text starting at *point_like point*. See *Shape.insert_text()*.

insert_textbox(*rect*, *buffer*, *fontsize=11*, *fontname='helv'*, *fontfile=None*, *idx=0*, *color=None*, *fill=None*, *render_mode=0*, *border_width=1*, *encoding=TEXT_ENCODING_LATIN*, *expandtabs=8*, *align=TEXT_ALIGN_LEFT*, *charwidths=None*, *rotate=0*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*, *overlay=True*)

- Changed in v1.18.4

PDF only: Insert text into the specified *rect_like* *rect*. See [Shape.insert_textbox\(\)](#).

draw_line(*p1*, *p2*, *color=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

- Changed in v1.18.4

PDF only: Draw a line from *p1* to *p2* (*point_like* s). See [Shape.draw_line\(\)](#).

draw_zigzag(*p1*, *p2*, *breadth=2*, *color=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

- Changed in v1.18.4

PDF only: Draw a zigzag line from *p1* to *p2* (*point_like* s). See [Shape.draw_zigzag\(\)](#).

draw_squiggle(*p1*, *p2*, *breadth=2*, *color=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

- Changed in v1.18.4

PDF only: Draw a squiggly (wavy, undulated) line from *p1* to *p2* (*point_like* s). See [Shape.draw_squiggle\(\)](#).

draw_circle(*center*, *radius*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

- Changed in v1.18.4

PDF only: Draw a circle around *center* (*point_like*) with a radius of *radius*. See [Shape.draw_circle\(\)](#).

draw_oval(*quad*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

- Changed in v1.18.4

PDF only: Draw an oval (ellipse) within the given *rect_like* or *quad_like*. See [Shape.draw_oval\(\)](#).

draw_sector(*center*, *point*, *angle*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *fullSector=True*, *overlay=True*, *closePath=False*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

- Changed in v1.18.4

PDF only: Draw a circular sector, optionally connecting the arc to the circle's center (like a piece of pie). See [Shape.draw_sector\(\)](#).

draw_polyline(*points*, *color=None*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *closePath=False*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

- Changed in v1.18.4

PDF only: Draw several connected lines defined by a sequence of *point_like* s. See [Shape.draw_polyline\(\)](#).

draw_bezier(*p1, p2, p3, p4, color=None, fill=None, width=1, dashes=None, lineCap=0, lineJoin=0, overlay=True, closePath=False, morph=None, stroke_opacity=1, fill_opacity=1, oc=0*)

- Changed in v1.18.4

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

draw_curve(*p1, p2, p3, color=None, fill=None, width=1, dashes=None, lineCap=0, lineJoin=0, overlay=True, closePath=False, morph=None, stroke_opacity=1, fill_opacity=1, oc=0*)

- Changed in v1.18.4

PDF only: This is a special case of [draw_bezier\(\)](#). See [Shape.draw_curve\(\)](#).

draw_rect(*rect, color=None, fill=None, width=1, dashes=None, lineCap=0, lineJoin=0, overlay=True, morph=None, stroke_opacity=1, fill_opacity=1, oc=0*)

- Changed in v1.18.4

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

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

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

draw_quad(*quad, color=None, fill=None, width=1, dashes=None, lineCap=0, lineJoin=0, overlay=True, morph=None, stroke_opacity=1, fill_opacity=1, oc=0*)

- Changed in v1.18.4

PDF only: Draw a quadrilateral. See [Shape.draw_quad\(\)](#).

insert_font(*fontname='helv', fontfile=None, fontbuffer=None, set_simple=False, encoding=TEXT_ENCODING_LATIN*)

PDF only: Add a new font to be used by text output methods and return its [xref](#). If not already present in the file, the font definition will be added. Supported are the built-in [Base14_Fonts](#) and the CJK fonts via “**reserved**” fontnames. Fonts can also be provided as a file path or a memory area containing the image of a font file.

Parameters

fontname (*str*) – The name by which this font shall be referenced when outputting text on this page. In general, you have a “free” choice here (but consult the [Adobe PDF References](#), page 16, section 7.3.5 for a formal description of building legal PDF names). However, if it matches one of the [Base14_Fonts](#) or one of the CJK fonts, *fontfile* and *fontbuffer* are ignored.

In other words, you cannot insert a font via *fontfile* / *fontbuffer* and also give it a reserved *fontname*.

Note: A reserved fontname can be specified in any mixture of upper or lower case and still match the right built-in font definition: fontnames “helv”, “Helv”, “HELV”, “Helvetica”, etc. all lead to the same font definition “Helvetica”. But from a [Page](#) perspective, these are **different references**. You can exploit this fact when using different *encoding* variants (Latin, Greek, Cyrillic) of the same font on a page.

Parameters

- **fontfile** (*str*) – a path to a font file. If used, *fontname* must be **different from all reserved names**.
- **fontbuffer** (*bytes/bytearray*) – the memory image of a font file. If used, *fontname* must be **different from all reserved names**. This parameter would typically be used with [Font.buffer](#) for fonts supported / available via [Font](#).
- **set_simple** (*int*) – applicable for *fontfile* / *fontbuffer* cases only: enforce treatment as a “simple” font, i.e. one that only uses character codes up to 255.
- **encoding** (*int*) – applicable for the “Helvetica”, “Courier” and “Times” sets of [Base14_Fonts](#) only. Select one of the available encodings Latin (0), Cyrillic (2) or Greek (1). Only use the default (0 = Latin) for “Symbol” and “ZapfDingBats”.

Rtype

int

Returnsthe [xref](#) of the installed font.

Note: Built-in fonts will not lead to the inclusion of a font file. So the resulting PDF file will remain small. However, your PDF viewer software is responsible for generating an appropriate appearance – and there **exist** differences on whether or how each one of them does this. This is especially true for the CJK fonts. But also Symbol and ZapfDingbats are incorrectly handled in some cases. Following are the **Font Names** and their correspondingly installed **Base Font** names:

Base-14 Fonts¹

Font Name	Installed Base Font	Comments
helv	Helvetica	normal
heit	Helvetica-Oblique	italic
hebo	Helvetica-Bold	bold
hebi	Helvetica-BoldOblique	bold-italic
cour	Courier	normal
coit	Courier-Oblique	italic
cobo	Courier-Bold	bold
cobi	Courier-BoldOblique	bold-italic
tiro	Times-Roman	normal
tiit	Times-Italic	italic
tibo	Times-Bold	bold
tibi	Times-BoldItalic	bold-italic
symb	Symbol	³
zadb	ZapfDingbats	³

CJK Fonts² (China, Japan, Korea)

¹ If your existing code already uses the installed base name as a font reference (as it was supported by PyMuPDF versions earlier than 1.14), this will continue to work.

³ Not all PDF readers display these fonts at all. Some others do, but use a wrong character spacing, etc.

² Not all PDF reader software (including internet browsers and office software) display all of these fonts. And if they do, the difference between the **serifed** and the **non-serifed** version may hardly be noticable. But serifed and non-serifed versions lead to different installed base fonts, thus providing an option to be displayable with your specific PDF viewer.

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)

insert_image(*rect*, *filename=None*, *pixmap=None*, *stream=None*, *mask=None*, *rotate=0*, *alpha=-1*, *oc=0*, *xref=0*, *keep_proportion=True*, *overlay=True*)

PDF only: Put an image inside the given rectangle. The image may already exist in the PDF or be taken from a pixmap, a file, or a memory area.

- Changed in v1.14.1: By default, the image keeps its aspect ratio.
- Changed in v1.14.13: The image is now always placed **centered** in the rectangle, i.e. the centers of image and rectangle are equal.
- Changed in v1.17.6: Insertion rectangle no longer needs to have a non-empty intersection with the page's [Page.croptbox](#)⁵.
- Changed in v1.18.13: Allow providing the image as the xref of an existing one.

Parameters

- **rect** (*rect_like*) – where to put the image. Must be finite and not empty.
- **filename** (*str*) – name of an image file (all formats supported by MuPDF – see [Supported Input Image Formats](#)).
- **stream** (*bytes, bytearray, io.BytesIO*) – image in memory (all formats supported by MuPDF – see [Supported Input Image Formats](#)).

Changed in v1.14.13: *io.BytesIO* is now also supported.

- **pixmap** (*Pixmap*) – a pixmap containing the image.
- **mask** (*bytes, bytearray, io.BytesIO*) – (*new in version v1.18.1*) image in memory – to be used as image mask (alpha values) for the base image. When specified, the base image must be provided as a filename or a stream – and must not be an image that already has a mask.
- **xref** (*int*) – (*New in v1.18.13*) the [xref](#) of an image already present in the PDF. If given, parameters *filename*, *pixmap*, *stream*, *alpha* and *mask* are ignored. The page will simply receive a reference to the existing image.
- **alpha** (*int*) – (*Changed in v1.19.3*) deprecated. No longer needed – ignored when given.
- **rotate** (*int*) – (*new in version v1.14.11*) rotate the image. Must be an integer multiple of 90 degrees. If you need a rotation by an arbitrary angle, consider converting the image to a PDF ([Document.convert_to_pdf\(\)](#)) first and then use [Page.show_pdf_page\(\)](#) instead.

⁵ The previous algorithm caused images to be **shrunk** to this intersection. Now the image can be anywhere on [Page.mediabox](#), potentially being invisible or only partially visible if the cropbox (representing the visible page part) is smaller.

- **oc** (*int*) – (new in v1.18.3) (*xref*) make image visibility dependent on this *OCG* or *OCMD*. Ignored after the first of multiple insertions. The property is stored with the generated PDF image object and therefore controls the image’s visibility throughout the PDF.
- **keep_proportion** (*bool*) – (new in version v1.14.11) maintain the aspect ratio of the image.

For a description of *overlay* see *Common Parameters*.

Changed in v1.18.13: Return xref of stored image.

Return type

int

Returns

The xref of the embedded image. This can be used as the *xref* argument for very significant performance boosts, if the image is inserted again.

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
>>> img_xref = 0                        # first execution embeds the image
>>> for page in doc:
>>>     img_xref = page.insert_image(rect, stream=img,
>>>                                 xref=img_xref, 2nd time reuses existing image
>>>                                 )
>>> doc.save(...)
```

Note:

1. The method detects multiple insertions of the same image (like in above example) and will store its data only on the first execution. This is even true (although less performant), if using the default *xref=0*.
 2. The method cannot detect if the same image had already been part of the file before opening it.
 3. You can use this method to provide a background or foreground image for the page, like a copyright or a watermark. Please remember, that watermarks require a transparent image if put in foreground ...
 4. 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. In addition, there exist effective ways to control the image size – even if transparency comes into play. Have a look at [this](#) section of the documentation.
 5. The image is stored in the PDF in its original quality. This may be much better than what you ever need for your display. Consider **decreasing the image size** before insertion – e.g. by using the *pixmap* option and then shrinking it or scaling it down (see *Pixmap* chapter). The PIL method *Image.thumbnail()* can also be used for that purpose. The file size savings can be very significant.
 6. Another efficient way to display the same image on multiple pages is another method: *show_pdf_page()*. Consult *Document.convert_to_pdf()* for how to obtain intermediary PDFs usable for that method. Demo script *fitz-logo.py* implements a fairly complete approach.
-

replace_image(*xref*, *filename=None*, *pixmap=None*, *stream=None*)

- New in v1.21.0

Replace the image at *xref* with another one.

Parameters

- **xref** (*int*) – the *xref* of the image.
- **filename** – the filename of the new image.
- **pixmap** – the *Pixmap* of the new image.
- **stream** – the memory area containing the new image.

Arguments *filename*, *pixmap*, *stream* have the same meaning as in *Page.insert_image()*, especially exactly one of these must be provided.

This is a **global replacement**: the new image will also be shown wherever the old one has been displayed throughout the file.

This method mainly exists for technical purposes. Typical uses include replacing large images by smaller versions, like a lower resolution, graylevel instead of colored, etc., or changing transparency.

delete_image(*xref*)

- New in v1.21.0

Delete the image at *xref*. This is slightly misleading: actually the image is being replaced with a small transparent *Pixmap* using above *Page.replace_image()*. The visible effect however is equivalent.

Parameters

- **xref** (*int*) – the *xref* of the image.

This is a **global replacement**: the image will disappear wherever the old one has been displayed throughout the file.

get_text(*opt*, *, *clip=None*, *flags=None*, *textpage=None*, *sort=False*)

- Changed in v1.19.0: added *textpage* parameter
- Changed in v1.19.1: added *sort* parameter
- Changed in v1.19.6: added new constants for defining default flags per method.

Retrieves the content of a page in a variety of formats. This is a wrapper for *TextPage* methods by choosing the output option as follows:

- “text” – *TextPage.extractTEXT()*, default
- “blocks” – *TextPage.extractBLOCKS()*
- “words” – *TextPage.extractWORDS()*
- “html” – *TextPage.extractHTML()*
- “xhtml” – *TextPage.extractXHTML()*
- “xml” – *TextPage.extractXML()*
- “dict” – *TextPage.extractDICT()*
- “json” – *TextPage.extractJSON()*
- “rawdct” – *TextPage.extractRAWDICT()*
- “rawjson” – *TextPage.extractRAWJSON()*

Parameters

- **opt** (*str*) – A string indicating the requested format, one of the above. A mixture of upper and lower case is supported.

Changed in v1.16.3 Values “words” and “blocks” are now also accepted.

- **clip** (*rect-like*) – (*new in v1.17.7*) restrict extracted text to this rectangle. If None, the full page is taken. Has **no effect** for options “html”, “xhtml” and “xml”.
- **flags** (*int*) – (*new in v1.16.2*) indicator bits to control whether to include images or how text should be handled with respect to white spaces and ligatures. See [Text Extraction Flags](#) for available indicators and [Text Extraction Flags Defaults](#) for default settings.
- **textpage** – (*new in v1.19.0*) use a previously created [TextPage](#). This reduces execution time **very significantly**: by more than 50% and up to 95%, depending on the extraction option. If specified, the ‘flags’ and ‘clip’ arguments are ignored, because they are textpage-only properties. If omitted, a new, temporary textpage will be created.
- **sort** (*bool*) – (*new in v1.19.1*) sort the output by vertical, then horizontal coordinates. In many cases, this should suffice to generate a “natural” reading order. Has no effect on (X)HTML and XML. Output option “**words**” sorts by (y1, x0) of the words’ bboxes. Similar is true for “blocks”, “dict”, “json”, “rawdict”, “rawjson”: they all are sorted by (y1, x0) of the resp. block bbox. If specified for “text”, then internally “blocks” is used.

Return type

str, list, dict

Returns

The page’s content as a string, a list or a dictionary. Refer to the corresponding [TextPage](#) method for details.

Note:

1. You can use this method as a **document conversion tool** from any supported document type (not only PDF!) to one of TEXT, HTML, XHTML or XML documents.
 2. The inclusion of text via the *clip* parameter is decided on a by-character level: (**changed in v1.18.2**) a character becomes part of the output, if its bbox is contained in *clip*. This **deviates** from the algorithm used in redaction annotations: a character will be **removed if its bbox intersects** any redaction annotation.
-

get_textbox(*rect, textpage=None*)

- New in v1.17.7
- Changed in v1.19.0: add **textpage** parameter

Retrieve the text contained in a rectangle.

Parameters

- **rect** (*rect-like*) – rect-like.
- **textpage** – a [TextPage](#) to use. If omitted, a new, temporary textpage will be created.

Returns

a string with interspersed linebreaks where necessary. Changed in v1.19.0: It is based on dedicated code. A typical use is checking the result of [Page.search_for\(\)](#):

```
>>> r1 = page.search_for("currency:")
>>> page.get_textbox(r1[0])
'Currency: '
>>>
```

get_textpage(*clip=None, flags=3*)

- New in v1.16.5
- Changed in v1.17.7: introduced *clip* parameter.

Create a [TextPage](#) for the page.

Parameters

- **flags** (*in*) – indicator bits controlling the content available for subsequent text extractions and searches – see the parameter of [Page.get_text\(\)](#).
- **clip** (*rect-like*) – (*new in v1.17.7*) restrict extracted text to this area.

Returns

[TextPage](#)

get_textpage_ocr(*flags=3, language='eng', dpi=72, full=False*)

- New in v1.19.0
- Changed in v1.19.1: support full and partial OCRing a page.

Create a [TextPage](#) for the page that includes OCRed text. MuPDF will invoke Tesseract-OCR if this method is used. Otherwise this is a normal [TextPage](#) object.

Parameters

- **flags** (*in*) – indicator bits controlling the content available for subsequent text extractions and searches – see the parameter of [Page.get_text\(\)](#).
- **language** (*str*) – the expected language(s). Use “+”-separated values if multiple languages are expected, “eng+spa” for English and Spanish.
- **dpi** (*int*) – the desired resolution in dots per inch. Influences recognition quality (and execution time).
- **full** (*bool*) – whether to OCR the full page, or just the displayed images.

Note: This method does **not** support a *clip* parameter – OCR will always happen for the complete page rectangle.

Returns

a [TextPage](#). Execution may be significantly longer than [Page.get_textpage\(\)](#).

For a full page OCR, **all text** will have the font “GlyphlessFont” from Tesseract. In case of partial OCR, normal text will keep its properties, and only text coming from images will have the GlyphlessFont.

Note: **OCRed text is only available** to PyMuPDF's text extractions and searches if their `textpage` parameter specifies the output of this method.

[This Jupyter notebook](#) walks through an example for using OCR textpages.

`get_drawings()`

- New in v1.18.0
- Changed in v1.18.17
- Changed in v1.19.0: add “seqno” key, remove “clippings” key
- Changed in v1.19.1: “color”/“fill” keys now always are either are RGB tuples or None. This resolves issues caused by exotic colorspace.
- Changed in v1.19.2: add an indicator for the “*orientation*” of the area covered by an “re” item.

Return the draw commands of the page. These are instructions which draw lines, rectangles, quadruples or curves, including properties like colors, transparency, line width and dashing, etc.

Returns

a list of dictionaries. Each dictionary item contains one or more single draw commands belonging together: they have the same properties (colors, dashing, etc.). This is called a “**path**” in PDF, but the method **works for all document types**.

The path dictionary has been designed to be compatible with class *Shape*. There are the following keys:

Key	Value
<code>closePath</code>	Same as the parameter in <i>Shape</i> .
<code>color</code>	Stroke color (see <i>Shape</i>).
<code>dashes</code>	Dashed line specification (see <i>Shape</i>).
<code>even_odd</code>	Fill colors of area overlaps – same as the parameter in <i>Shape</i> .
<code>fill</code>	Fill color (see <i>Shape</i>).
<code>items</code>	List of draw commands: lines, rectangles, quads or curves.
<code>lineCap</code>	Number 3-tuple, use its max value on output with <i>Shape</i> .
<code>lineJoin</code>	Same as the parameter in <i>Shape</i> .
<code>fill_opacity</code>	(new in v1.18.17) fill color transparency (see <i>Shape</i>).
<code>stroke_opacity</code>	(new in v1.18.17) stroke color transparency (see <i>Shape</i>).
<code>rect</code>	Page area covered by this path. Information only.
<code>seqno</code>	(new in v1.19.0) command number when building page appearance
<code>type</code>	(new in v1.18.17) type of this path.
<code>width</code>	Stroke line width (see <i>Shape</i>).

- (Changed in v1.18.17) Key “opacity” has been replaced by the new keys “fill_opacity” and “stroke_opacity”. This is now compatible with the corresponding parameters of *Shape.finish()*.

Key “type” takes one of the following values:

- “f” – this is a *fill-only* path. Only key-values relevant for this operation have a meaning, irrelevant ones have been added with default values for backward compatibility: “color”, “lineCap”, “lineJoin”, “width”, “closePath”, “dashes” and should be ignored.

- “s” – this is a *stroke-only* path. Similar to previous, key “fill” is present with value `None`.
- “fs” – this is a path performing combined *fill* and *stroke* operations.

Each item in `path["items"]` is one of the following:

- (“l”, p1, p2) - a line from p1 to p2 (*Point* objects).
- (“c”, p1, p2, p3, p4) - cubic Bézier curve **from p1 to p4** (p2 and p3 are the control points). All objects are of type *Point*.
- (“re”, rect, orientation) - a *Rect*. *Changed in v1.18.17:* Multiple rectangles within the same path are now detected. *Changed in v1.19.2:* added integer **orientation** which is 1 resp. -1 indicating whether the enclosed area is rotated left (1 = anti-clockwise), or resp. right⁷.
- (“qu”, quad) - a *Quad*. *New in v1.18.17, changed in v1.19.2:* 3 or 4 consecutive lines are detected to actually represent a *Quad*.

Note: Starting with v1.19.2, quads and rectangles are more reliably recognized as such.

Using class *Shape*, you should be able to recreate the original drawings on a separate (PDF) page with high fidelity under normal, not too sophisticated circumstances. Please see the following comments on restrictions. A coding draft can be found in section “Extractings Drawings” of chapter FAQ.

Note:

- The visual appearance of a page may have been designed in a very complex way. For example in PDF:
 - Layers (Optional Content Groups) can control the visibility of items (drawings and other objects) depending on whatever condition: for example showing or suppressing a watermark depending on the current output device (screen, paper, ...), or option-based inclusion / omission of details in a technical document, and so on. Effects like these are ignored by the method – it will **unconditionally return all paths**.
 - Clipping paths are ignored by the method.
 - Shadings (gradient colorization effects) are not supported.
 - When a viewer software builds a page’s appearance, it will sequentially walk through a list of commands (in PDF, those are stored in the `/Contents` object), containing instructions like “draw this path, show this image, paint this text, etc.”. The key “seqno” (new in v1.19.0) is the command number, that draws this path. You can use it to determine if objects cover other objects on the page. For example, the rectangle of a “fill” path will cover objects drawn earlier – i.e. having a smaller “seqno” – if the rectangles overlap. Please also see *Page.get_bboxlog()* and *Page.get_texttrace()*.
-

Note: The method is now based on the output of *Page.get_cdrawings()* – which is much faster, but requires somewhat more attention processing its output.

⁷ In PDF, an area enclosed by some lines or curves can have a property called “orientation”. This is significant for switching on or off the fill color of that area when there exist multiple area overlaps - see discussion in method *Shape.finish()* using the “non-zero winding number” rule. While orientation of curves, quads, triangles and other shapes enclosed by lines always was detectable, this has been impossible for “re” (rectangle) items in the past. Adding the orientation parameter now delivers the missing information.

get_cdrawings()

- New in v1.18.17
- Changed in v1.19.0: removed “clippings” key, added “seqno” key.
- Changed in v1.19.1: always generate RGB color tuples.

Extract the drawing paths on the page. Apart from following technical differences, functionally equivalent to [Page.get_drawings\(\)](#), but much faster (factor 3 or more):

- Every path type only contains the relevant keys, e.g. a stroke path has no “fill” color key. See comment in method [Page.get_drawings\(\)](#).
- Coordinates are given as [point_like](#), [rect_like](#) and [quad_like](#) tuples – not as [Point](#), [Rect](#), [Quad](#) objects.

Note: If performance is a concern (e.g. because your page has tens of thousands of drawings), consider using this method: Compared to versions earlier than 1.18.17, you should see much shorter response times. We have seen pages that required 2 seconds then, now only need 200 ms with this method.

get_fonts(full=False)

PDF only: Return a list of fonts referenced by the page. Wrapper for [Document.get_page_fonts\(\)](#).

get_images(full=False)

PDF only: Return a list of images referenced by the page. Wrapper for [Document.get_page_images\(\)](#).

get_image_info(hashes=False, xrefs=False)

- *New in v1.18.11*
- *Changed in v1.18.13:* added image MD5 hashcode computation and [xref](#) search.

Return a list of meta information dictionaries for all images shown on the page. This works for all document types. Technically, this is a subset of the dictionary output of [Page.get_text\(\)](#): the image binary content and any text on the page are ignored.

Parameters

- **hashes** (*bool*) – *New in v1.18.13:* Compute the MD5 hashcode for each encountered image, which allows identifying image duplicates. This adds the key “digest” to the output, whose value is a 16 byte bytes object.
- **xrefs** (*bool*) – *New in v1.18.13: PDF only.* Try to find the [xref](#) for each image. Implies **hashes=True**. Adds the “xref” key to the dictionary. If not found, the value is 0, which means, the image is either “inline” or otherwise undetectable. Please note that this option has an extended response time, because the MD5 hashcode will be computed at least two times for each image with an xref.

Return type

list[dict]

Returns

A list of dictionaries. This includes information for **exactly those** images, that are shown on the page – including “*inline images*”. In contrast to images included in [Page.get_text\(\)](#), image **binary content** is not loaded, which drastically reduces memory usage. The dictionary layout is similar to that of image blocks in [page.get_text\("dict"\)](#).

Key	Value
number	block number (<i>int</i>)
bbox	image bbox on page, <i>rect_like</i>
width	original image width (<i>int</i>)
height	original image height (<i>int</i>)
cs-name	colorspace name (<i>str</i>)
colorspace	colorspace.n (<i>int</i>)
xres	resolution in x-direction (<i>int</i>)
yres	resolution in y-direction (<i>int</i>)
bpc	bits per component (<i>int</i>)
size	storage occupied by image (<i>int</i>)
digest	MD5 hashcode (<i>bytes</i>), if <i>hashes</i> is true
xref	image <i>xref</i> or 0, if <i>xrefs</i> is true
transform	matrix transforming image rect to bbox, <i>matrix_like</i>

Multiple occurrences of the same image are always reported. You can detect duplicates by comparing their *digest* values.

`get_xobjects()`

PDF only: Return a list of Form XObjects referenced by the page. Wrapper for *Document.get_page_xobjects()*.

`get_image_rects(item, transform=False)`

New in v1.18.13

PDF only: Return boundary boxes and transformation matrices of an embedded image. This is an improved version of *Page.get_image_bbox()* with the following differences:

- There is no restriction on **how** the image is invoked (by the page or one of its Form XObjects). The result is always complete and correct.
- The result is a list of *Rect* or (*Rect*, *Matrix*) objects – depending on *transform*. Each list item represents one location of the image on the page. Multiple occurrences might not be detectable by *Page.get_image_bbox()*.
- The method invokes *Page.get_image_info()* with *xrefs=True* and therefore has a noticeably longer response time than *Page.get_image_bbox()*.

Parameters

- **item** (*list*, *str*, *int*) – an item of the list *Page.get_images()*, or the reference **name** entry of such an item (item[7]), or the image *xref*.
- **transform** (*bool*) – also return the matrix used to transform the image rectangle to the bbox on the page. If true, then tuples (bbox, matrix) are returned.

Return type

list

Returns

Boundary boxes and respective transformation matrices for each image occurrence on the page. If the item is not on the page, an empty list [] is returned.

`get_image_bbox(item, transform=False)`

- Changed in v1.18.11: return image transformation matrix

PDF only: Return boundary box and transformation matrix of an embedded image.

Parameters

- **item** (*list*, *str*) – an item of the list [Page.get_images\(\)](#) with *full=True* specified, or the reference **name** entry of such an item, which is item[-3] (or item[7] respectively).
- **transform** (*bool*) – (new in v1.18.11) also return the matrix used to transform the image rectangle to the bbox on the page. Default is just the bbox. If true, then a tuple (bbox, matrix) is returned.

Return type

[Rect](#) or ([Rect](#), [Matrix](#))

Returns

the boundary box of the image – optionally also its transformation matrix.

- (Changed in v1.16.7) – If the page in fact does not display this image, an infinite rectangle is returned now. In previous versions, an exception was raised. Formally invalid parameters still raise exceptions.
- (Changed in v1.17.0) – Only images referenced directly by the page are considered. This means that images occurring in embedded PDF pages are ignored and an exception is raised.
- (Changed in v1.18.5) – Removed the restriction introduced in v1.17.0: any item of the page’s image list may be specified.
- (Changed in v1.18.11) – Partially re-instated a restriction: only those images are considered, that are either directly referenced by the page or by a Form XObject directly referenced by the page.
- (Changed in v1.18.11) – Optionally also return the transformation matrix together with the bbox as the tuple (bbox, transform).

Note:

1. Be aware that [Page.get_images\(\)](#) may contain “dead” entries i.e. images, which the page **does not display**. This is no error, but intended by the PDF creator. No exception will be raised in this case, but an infinite rectangle is returned. You can avoid this from happening by executing [Page.clean_contents\(\)](#) before this method.
 2. The image’s “transformation matrix” is defined as the matrix, for which the expression `bbox / transform == fitz.Rect(0, 0, 1, 1)` is true, lookup details here: [Image Transformation Matrix](#).
-

get_svg_image(*matrix=fitz.Identity*, *text_as_path=True*)

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

Parameters

- **matrix** (*matrix_like*) – a matrix, default is [Identity](#).
- **text_as_path** (*bool*) – (new in v1.17.5) – controls how text is represented. *True* outputs each character as a series of elementary draw commands, which leads to a more precise text display in browsers, but a **very much larger** output for text-oriented pages. Display quality for *False* relies on the presence of the referenced fonts on the current system. For missing fonts, the internet browser will fall back

to some default – leading to unpleasant appearances. Choose *False* if you want to parse the text of the SVG.

Returns

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

Note: In case of a PDF, you can circumvent the “full page image only” restriction by modifying the page’s CropBox before using the method.

```
get_pixmap(*, matrix=fitz.Identity, dpi=None, colorspace=fitz.csRGB, clip=None, alpha=False,
           annots=True)
```

- Changed in v1.19.2: added support of parameter `dpi`.

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

All parameters are *keyword-only*.

Parameters

- **matrix** (*matrix_like*) – default is *Identity*.
- **dpi** (*int*) – (new in v1.19.2) desired resolution in x and y direction. If not *None*, the “matrix” parameter is ignored.
- **colorspace** (*str* or *Colorspace*) – The desired colorspace, one of “GRAY”, “RGB” or “CMYK” (case insensitive). Or specify a *Colorspace*, ie. one of the predefined ones: *csGRAY*, *csRGB* or *csCMYK*.
- **clip** (*irect_like*) – restrict rendering to the intersection of this area with the page’s rectangle.
- **alpha** (*bool*) – whether to add an alpha channel. Always accept the default *False* if you do not really need transparency. This will save a lot of memory (25% in case of RGB ... and pixmaps are typically **large!**), and also processing time. Also note an **important difference** in how the image will be rendered: with *True* the pixmap’s samples area will be pre-cleared with *0x00*. This results in **transparent** areas where the page is empty. With *False* the pixmap’s samples will be pre-cleared with *0xff*. This results in **white** where the page has nothing to show.

Changed in v1.14.17

The default alpha value is now *False*.

– Generated with *alpha=True*



– Generated with *alpha=False*



- **annots** (*bool*) – (new in vrsion 1.16.0) whether to also render annotations or to suppress them. You can create pixmaps for annotations separately.

Return type*Pixmap***Returns**

Pixmap of the page. For fine-controlling the generated image, the by far most important parameter is **matrix**. E.g. you can increase or decrease the image resolution by using **Matrix(xzoom, yzoom)**. If `zoom > 1`, you will get a higher resolution: `zoom=2` will double the number of pixels in that direction and thus generate a 2 times larger image. Non-positive values will flip horizontally, resp. vertically. Similarly, matrices also let you rotate or shear, and you can combine effects via e.g. matrix multiplication. See the [Matrix](#) section to learn more.

Note: The method will respect any page rotation and will not exceed the intersection of `clip` and [Page.cropbox](#). If you need the page's mediabox (and if this is a different rectangle), you can use a snippet like the following to achieve this:

```
In [1]: import fitz
In [2]: doc=fitz.open("demo1.pdf")
In [3]: page=doc[0]
In [4]: rotation = page.rotation
In [5]: cropbox = page.cropbox
In [6]: page.set_cropbox(page.mediabox)
In [7]: page.set_rotation(0)
In [8]: pix = page.get_pixmap()
In [9]: page.set_cropbox(cropbox)
In [10]: if rotation != 0:
...:     page.set_rotation(rotation)
...:
In [11]:
```

annot_names()

- New in v1.16.10

PDF only: return a list of the names of annotations, widgets and links. Technically, these are the */NM* values of every PDF object found in the page's */Annots* array.

Return type

list

annot_xrefs()

- New in v1.17.1

PDF only: return a list of the `:data`xref`` numbers of annotations, widgets and links – technically of all entries found in the page’s `/Annots` array.

Return type

list

Returns

a list of items (*xref*, *type*) where *type* is the annotation type. Use the type to tell apart links, fields and annotations, see [Annotation Types](#).

load_annot(*ident*)

- New in v1.17.1

PDF only: return the annotation identified by *ident*. This may be its unique name (PDF `/NM` key), or its [xref](#).

Parameters

ident (*str*, *int*) – the annotation name or xref.

Return type

[Annot](#)

Returns

the annotation or *None*.

Note: Methods [Page.annot_names\(\)](#), [Page.annot_xrefs\(\)](#) provide lists of names or xrefs, respectively, from where an item may be picked and loaded via this method.

load_widget(*xref*)

- New in v1.19.6

PDF only: return the field identified by *xref*.

Parameters

xref (*int*) – the field’s xref.

Return type

[Widget](#)

Returns

the field or *None*.

Note: This is similar to the analogous method [Page.load_annot\(\)](#) – except that here only the xref is supported as identifier.

load_links()

Return the first link on a page. Synonym of property [first_link](#).

Return type

[Link](#)

Returns

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

set_rotation(*rotate*)

PDF only: Set the rotation of the page.

Parameters

rotate (*int*) – An integer specifying the required rotation in degrees. Must be an integer multiple of 90. Values will be converted to one of 0, 90, 180, 270.

show_pdf_page(*rect*, *docsrc*, *pno*=0, *keep_proportion*=True, *overlay*=True, *oc*=0, *rotate*=0, *clip*=None)

- Changed in v1.14.11: Parameter *reuse_xref* has been deprecated. Position the source rectangle centered in target rectangle. Any rotation angle is now supported.
- Changed in v1.18.3: New parameter *oc*.

PDF only: Display a page of another PDF as a **vector image** (otherwise similar to [Page.insert_image\(\)](#)). This is a multi-purpose method. For example, you can use it to

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

Parameters

- **rect** (*rect_like*) – where to place the image on current page. Must be finite and its intersection with the page must not be empty.
- **docsrc** (*Document*) – source PDF document containing the page. Must be a different document object, but may be the same file.
- **pno** (*int*) – page number (0-based, in $-\infty < \text{pno} < \text{docsrc.page_count}$) to be shown.
- **keep_proportion** (*bool*) – whether to maintain the width-height-ratio (default). If false, all 4 corners are always positioned on the border of the target rectangle – whatever the rotation value. In general, this will deliver distorted and /or non-rectangular images.
- **overlay** (*bool*) – put image in foreground (default) or background.
- **oc** (*int*) – (new in v1.18.3) (*xref*) make visibility dependent on this OCG (optional content group).
- **rotate** (*float*) – (new in v1.14.10) show the source rectangle rotated by some angle. *Changed in v1.14.11: Any angle is now supported.*
- **clip** (*rect_like*) – choose which part of the source page to show. Default is the full page, else must be finite and its intersection with the source page must not be empty.

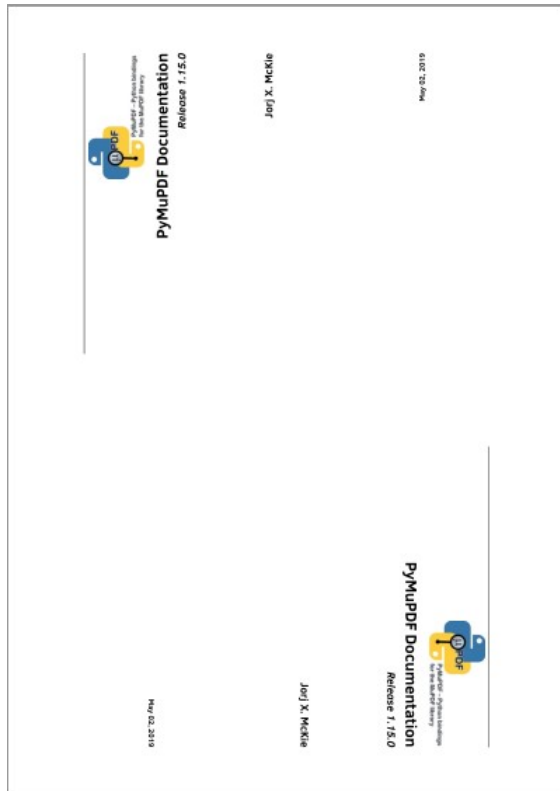
Note: In contrast to method [Document.insert_pdf\(\)](#), this method does not copy annotations, widgets or links, so these are not included in the target⁶. But all its **other resources (text, images, fonts, etc.)** will

⁶ If you need to also see annotations or fields in the target page, you can try and convert the source PDF to another PDF using [Document.convert_to_pdf\(\)](#). The underlying MuPDF function of that method will convert these objects to normal page content. Then use [Page.show_pdf_page\(\)](#) with the converted PDF page.

be imported into the current PDF. They will therefore appear in text extractions and in `get_fonts()` and `get_images()` lists – even if they are not contained in the visible area given by `clip`.

Example: Show the same source page, rotated by 90 and by -90 degrees:

```
>>> doc = fitz.open() # new empty PDF
>>> page=doc.new_page() # new page in A4 format
>>>
>>> # upper half page
>>> r1 = fitz.Rect(0, 0, page.rect.width, page.rect.height/2)
>>>
>>> # lower half page
>>> r2 = r1 + (0, page.rect.height/2, 0, page.rect.height/2)
>>>
>>> src = fitz.open("PyMuPDF.pdf") # show page 0 of this
>>>
>>> page.show_pdf_page(r1, src, 0, rotate=90)
>>> page.show_pdf_page(r2, src, 0, rotate=-90)
>>> doc.save("show.pdf")
```



`new_shape()`

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.

```
search_for(needle, *, clip=clip, quads=False, flags=TEXT_DEHYPHENATE |  
           TEXT_PRESERVE_WHITESPACE | TEXT_PRESERVE_LIGATURES, textpage=None)
```

- Changed in v1.18.2: added `clip` parameter. Remove `hit_max` parameter. Add default “dehyphenate”.
- Changed in v1.19.0: added `textpage` parameter.

Search for *needle* on a page. Wrapper for [TextPage.search\(\)](#).

Parameters

- **needle** (*str*) – Text to search for. May contain spaces. Upper / lower case is ignored, but only works for ASCII characters: For example, “COMPÉTENCES” will not be found if needle is “compétences” – “compÉtences” however will. Similar is true for German umlauts and the like.
- **clip** (*rect_like*) – (New in v1.18.2) only search within this area.
- **quads** (*bool*) – Return object type [Quad](#) instead of [Rect](#).
- **flags** (*int*) – Control the data extracted by the underlying [TextPage](#). By default, ligatures and white spaces are kept, and hyphenation⁸ is detected.
- **textpage** – (new in v1.19.0) use a previously created [TextPage](#). This reduces execution time **significantly**. If specified, the ‘flags’ and ‘clip’ arguments are ignored. If omitted, a temporary textpage will be created.

Return type

list

Returns

A list of [Rect](#) or [Quad](#) objects, each of which – **normally!** – surrounds one occurrence of *needle*. **However:** if parts of *needle* occur on more than one line, then a separate item is generated for each these parts. So, if `needle = "search string"`, two rectangles may be generated.

Changes in v1.18.2:

- There no longer is a limit on the list length (removal of the `hit_max` parameter).
- If a word is **hyphenated** at a line break, it will still be found. E.g. the needle “method” will be found even if hyphenated as “meth-od” at a line break, and two rectangles will be returned: one surrounding “meth” (without the hyphen) and another one surrounding “od”.

Note: The method supports multi-line text marker annotations: you can use the full returned list as **one single** parameter for creating the annotation.

Caution:

- There is a tricky aspect: the search logic regards **contiguous multiple occurrences** of *needle* as one: assuming *needle* is “abc”, and the page contains “abc” and “abcabc”, then only **two** rectangles will be returned, one for “abc”, and a second one for “abcabc”.

⁸ Hyphenation detection simply means that if the last character of a line is “-”, it will be assumed to be a continuation character. That character will not be found by text searching with its default flag setting. Please take note, that a MuPDF *line* may not always be what you expect: words separated by overly large gaps (e.g. caused by text justification) may constitute separate MuPDF lines. If then any of these words ends with a hyphen, it will only be found by text searching if hyphenation is switched off.

- You can always use `Page.get_textbox()` to check what text actually is being surrounded by each rectangle.

Note: A feature repeatedly asked for is supporting **regular expressions** when specifying the "needle" string: **There is no way to do this.** If you need something in that direction, first extract text in the desired format and then subselect the result by matching with some regex pattern. Here is an example for matching words:

```
>>> pattern = re.compile(r"...") # the regex pattern
>>> words = page.get_text("words") # extract words on page
>>> matches = [w for w in words if pattern.search(w[4])]
```

The matches list will contain the words matching the given pattern. In the same way you can select `span["text"]` from the output of `page.get_text("dict")`.

set_mediabox(*r*)

- New in v1.16.13
- Changed in v1.19.4: remove all other rectangle definitions.

PDF only: Change the physical page dimension by setting *mediabox* in the page's object definition.

Parameters

r (*rect-like*) – the new *mediabox* value.

Note: This method also removes the page's other (optional) rectangles (*cropbox*, *ArtBox*, *TrimBox* and *Bleedbox*) to prevent inconsistent situations. This will cause those to assume their default values.

Caution: For non-empty pages this may have undesired effects, because the location of all content depends on this value and will therefore change position or even disappear.

set_cropbox(*r*)

PDF only: change the visible part of the page.

Parameters

r (*rect-like*) – the new visible area of the page. Note that this **must** be specified in **unrotated coordinates**, not empty, nor infinite and be completely contained in the *Page.mediabox*.

After execution (**if the page is not rotated**), *Page.rect* will equal this rectangle, but be shifted to the top-left position (0, 0) if necessary. Example session:

```
>>> page = doc.new_page()
>>> 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)
>>>
```

(continues on next page)

(continued from previous page)

```

>>> # now set cropbox to a part of the page
>>> page.set_cropbox(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 CropBox change
>>> # either set it to MediaBox
>>> page.set_cropbox(page.mediabox)
>>> # or 'refresh' MediaBox: will remove all other rectangles
>>> page.set_mediabox(page.mediabox)

```

set_artbox(*r*)**set_bleedbox(*r*)****set_trimbox(*r*)**

- New in v1.19.4

PDF only: Set the resp. rectangle in the page object. For the meaning of these objects see [Adobe PDF References](#), page 77. Parameter and restrictions are the same as for [Page.set_cropbox\(\)](#).

rotation

Contains the rotation of the page in degrees (always 0 for non-PDF types).

Type*int***cropbox_position**

Contains the top-left point of the page's /CropBox for a PDF, otherwise *Point(0, 0)*.

Type*Point***cropbox**

The page's /CropBox for a PDF. Always the **unrotated** page rectangle is returned. For a non-PDF this will always equal the page rectangle.

Note: In PDF, the relationship between /MediaBox, /CropBox and page rectangle may sometimes be confusing, please do lookup the glossary for [MediaBox](#).

Type*Rect***artbox****bleedbox**

trimbox

The page's /ArtBox, /BleedBox, /TrimBox, respectively. If not provided, defaulting to *Page.cropbox*.

Type

Rect

mediabox_size

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

Type

Point

mediabox

The page's *mediabox* for a PDF, otherwise *Page.rect*.

Type

Rect

Note: For most PDF documents and for **all other document types**, `page.rect == page.cropbox == page.mediabox` is true. However, for some PDFs the visible page is a true subset of *mediabox*. Also, if the page is rotated, its *Page.rect* may not equal *Page.cropbox*. In these cases the above attributes help to correctly locate page elements.

transformation_matrix

This matrix translates coordinates from the PDF space to the MuPDF space. For example, in PDF /Rect [x0 y0 x1 y1] the pair (x0, y0) specifies the **bottom-left** point of the rectangle – in contrast to MuPDF's system, where (x0, y0) specify top-left. Multiplying the PDF coordinates with this matrix will deliver the (Py-) MuPDF rectangle version. Obviously, the inverse matrix will again yield the PDF rectangle.

Type

Matrix

rotation_matrix**derotation_matrix**

These matrices may be used for dealing with rotated PDF pages. When adding / inserting anything to a PDF page, the coordinates of the **unrotated** page are always used. These matrices help translating between the two states. Example: if a page is rotated by 90 degrees – what would then be the coordinates of the top-left Point(0, 0) of an A4 page?

```
>>> page.set_rotation(90) # rotate an ISO A4 page
>>> page.rect
Rect(0.0, 0.0, 842.0, 595.0)
>>> p = fitz.Point(0, 0) # where did top-left point land?
>>> p * page.rotation_matrix
Point(842.0, 0.0)
>>>
```

Type

Matrix

first_link

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

Type

Link

first_annot

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

Type

Annot

first_widget

Contains the first *Widget* of a page (or *None*).

Type

Widget

number

The page number.

Type

int

parent

The owning document object.

Type

Document

rect

Contains the rectangle of the page. Same as result of *Page*.*bound()*.

Type

Rect

xref

The page's PDF *xref*. Zero if not a PDF.

Type

Rect

6.14.2 Description of *get_links()* Entries

Each entry of the *Page*.*get_links()* 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. You should make sure to start this string with an unambiguous substring, that classifies the subtype of the URL, like "http://", "https://", "file://", "ftp://", "mailto:", etc. Otherwise your browser will try to interpret the text and come to unwanted / unexpected conclusions about the intended URL type.
- *xref*: an integer specifying the PDF *xref* of the link object. Do not change this entry in any way. Required for link deletion and update, otherwise ignored. For non-PDF documents, this entry contains *-1*. It is also *-1* for **all** entries in the *get_links()* list, if **any** of the links is not supported by MuPDF - see the note below.

6.14.3 Notes on Supporting Links

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

6.14.3.1 Reading (pertains to method *get_links()* and the *first_link* 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 *get_links()* list will contain this information as the *to* value.

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

6.14.3.2 Writing

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

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

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

Example: How to insert a link pointing to another page in the same document

1. Determine the rectangle on the current page, where the link should be placed. This may be the bbox of an image or some text.
2. Determine the target page number ("pno", 0-based) and a *Point* on it, where the link should be directed to.
3. Create a dictionary `d = {"kind": fitz.LINK_GOTO, "page": pno, "from": bbox, "to": point}`.
4. Execute `page.insert_link(d)`.

6.14.4 Homologous Methods of Document and Page

This is an overview of homologous methods on the *Document* and on the *Page* level.

Document Level	Page Level
<i>Document.get_page_fonts(pno)</i>	<i>Page.get_fonts()</i>
<i>Document.get_page_images(pno)</i>	<i>Page.get_images()</i>
<i>Document.get_page_pixmap(pno, ...)</i>	<i>Page.get_pixmap()</i>
<i>Document.get_page_text(pno, ...)</i>	<i>Page.get_text()</i>
<i>Document.search_page_for(pno, ...)</i>	<i>Page.search_for()</i>

The page number “pno” is a 0-based integer $-\infty < \text{pno} < \text{page_count}$.

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.get_fonts()* is a wrapper the other way round and defined as follows: *page.get_fonts == page.parent.get_page_fonts(page.number)*.

6.15 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.get_pixmap()*)
2. empty, based on *Colorspace* and *IRect* information
3. from a file
4. from an in-memory image
5. from a memory area of plain pixels
6. from an image inside a PDF document
7. as a copy of another pixmap

Note: A number of image formats is supported as input for points 3. and 4. above. See section *Supported Input Image Formats*.

Have a look at the FAQ section to see some pixmap usage “at work”.

Method / Attribute	Short Description
<i>Pixmap.clear_with()</i>	clear parts of the pixmap
<i>Pixmap.color_count()</i>	determine used colors

continues on next page

Table 4 – continued from previous page

Method / Attribute	Short Description
<code>Pixmap.color_topusage()</code>	determine share of top used color
<code>Pixmap.copy()</code>	copy parts of another pixmap
<code>Pixmap.gamma_with()</code>	apply a gamma factor to the pixmap
<code>Pixmap.invert_irect()</code>	invert the pixels of a given area
<code>Pixmap.pdfocr_save()</code>	save the pixmap as an OCR'd 1-page PDF
<code>Pixmap.pdfocr_tobytes()</code>	save the pixmap as an OCR'd 1-page PDF
<code>Pixmap.pil_save()</code>	save as image using pillow
<code>Pixmap.pil_tobytes()</code>	write to bytes object using pillow
<code>Pixmap.pixel()</code>	return the value of a pixel
<code>Pixmap.save()</code>	save the pixmap in a variety of formats
<code>Pixmap.set_alpha()</code>	set alpha values
<code>Pixmap.set_dpi()</code>	set the image resolution
<code>Pixmap.set_origin()</code>	set pixmap x,y values
<code>Pixmap.set_pixel()</code>	set color and alpha of a pixel
<code>Pixmap.set_rect()</code>	set color and alpha of all pixels in a rectangle
<code>Pixmap.shrink()</code>	reduce size keeping proportions
<code>Pixmap.tint_with()</code>	tint the pixmap with a color
<code>Pixmap.tobytes()</code>	return a memory area in a variety of formats
<code>Pixmap.warp()</code>	return a pixmap made from a quad inside
<code>Pixmap.alpha</code>	transparency indicator
<code>Pixmap.colorspace</code>	pixmap's <i>Colorspace</i>
<code>Pixmap.digest</code>	MD5 hashcode of the pixmap
<code>Pixmap.height</code>	pixmap height
<code>Pixmap.interpolate</code>	interpolation method indicator
<code>Pixmap.is_monochrome</code>	check if only black and white occur
<code>Pixmap.is_unicolor</code>	check if only one color occurs
<code>Pixmap.irect</code>	<i>IRect</i> of the pixmap
<code>Pixmap.n</code>	bytes per pixel
<code>Pixmap.samples_mv</code>	memoryview of pixel area
<code>Pixmap.samples_ptr</code>	Python pointer to pixel area
<code>Pixmap.samples</code>	bytes copy of pixel area
<code>Pixmap.size</code>	pixmap's total length
<code>Pixmap.stride</code>	size of one image row
<code>Pixmap.width</code>	pixmap width
<code>Pixmap.x</code>	X-coordinate of top-left corner
<code>Pixmap.xres</code>	resolution in X-direction
<code>Pixmap.y</code>	Y-coordinate of top-left corner
<code>Pixmap.yres</code>	resolution in Y-direction

Class API

class Pixmap

`__init__(self, colorspace, irect, alpha)`

New empty pixmap: Create an empty pixmap of size and origin given by the rectangle. So, `irect.top_left` designates the top left corner of the pixmap, and its width and height are `irect.width` resp. `irect.height`. Note that the image area is **not initialized** and will contain crap data – use eg. `clear_with()` or `set_rect()` to be sure.

Parameters

- **colorspace** (*Colorspace*) – colorspace.

- **irect** (*irect_like*) – The 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, mask*)

- New in v1.18.18

Copy and add image mask: Copy *source* pixmap, add an alpha channel with transparency data from a mask pixmap.

Parameters

- **source** (*Pixmap*) – pixmap without alpha channel.
- **mask** (*Pixmap*) – a mask pixmap. Must be a grayscale pixmap.

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

Copy and scale: Copy *source* pixmap, scaling new width and height values – the image will appear stretched or shrunk accordingly. Supports partial copying. The source colorspace may be *None*.

Parameters

- **source** (*Pixmap*) – the source pixmap.
- **width** (*float*) – desired target width.
- **height** (*float*) – desired target height.
- **clip** (*irect_like*) – restrict the resulting pixmap to this region of the **scaled** pixmap.

Note: If width or height do not *represent* integers (i.e. `value.is_integer() != True`), then the resulting pixmap **will have an alpha channel**.

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

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

Parameters

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

Note: A typical use includes separation of color and transparency bytes in separate pixmaps. Some applications require this like e.g. `wx.Bitmap.FromBufferAndAlpha()` of `wxPython`:

```
>>> # 'pix' is an RGBA pixmap
>>> pixcolors = fitz.Pixmap(pix, 0)    # extract the RGB part (drop alpha)
>>> pixalpha = fitz.Pixmap(None, pix)  # extract the alpha part
>>> bm = wx.Bitmap.FromBufferAndAlpha(pix.width, pix.height, pixcolors.samples,
→ pixalpha.samples)
```

__init__(self, filename)

From a file: Create a pixmap from *filename*. All properties are inferred from the input. The origin of the resulting pixmap is (0, 0).

Parameters

filename (*str*) – Path of the image file.

__init__(self, stream)

From memory: Create a pixmap from a memory area. All properties are inferred from the input. The origin of the resulting pixmap is (0, 0).

Parameters

stream (*bytes, bytearray, BytesIO*) – Data containing a complete, valid image. Could have been created by e.g. `stream = bytearray(open('image.file', 'rb').read())`. Type *bytes* is supported in **Python 3 only**, because `bytes == str` in Python 2 and the method will interpret the stream as a filename.

Changed in version 1.14.13: *io.BytesIO* is now also supported.

__init__(self, colorspace, width, height, samples, alpha)

From plain pixels: Create a pixmap from *samples*. Each pixel must be represented by a number of bytes as controlled by the *colorspace* and *alpha* parameters. The origin of the resulting pixmap is (0, 0). This method is useful when raw image data are provided by some other program – see FAQ.

Parameters

- **colorspace** (*Colorspace*) – Colorspace of image.
- **width** (*int*) – image width
- **height** (*int*) – image height
- **samples** (*bytes, bytearray, BytesIO*) – an area containing all pixels of the image. Must include alpha values if specified.

Changed in version 1.14.13: (1) *io.BytesIO* can now also be used. (2) Data are now **copied** to the pixmap, so may safely be deleted or become unavailable.

- **alpha** (*bool*) – whether a transparency channel is included.

Note:

1. The following equation **must be true**: $(colorspace.n + alpha) * width * height == len(samples)$.
2. Starting with version 1.14.13, the samples data are **copied** to the pixmap.

__init__(self, doc, xref)

From a PDF image: Create a pixmap from an image **contained in PDF** *doc* identified by its *xref*. All pixmap properties are set by the image. Have a look at [extract-img1.py](#) and [extract-img2.py](#) to see how this can be used to recover all of a PDF's images.

Parameters

- **doc** (*Document*) – an opened **PDF** document.
- **xref** (*int*) – the *xref* of an image object. For example, you can make a list of images used on a particular page with `Document.get_page_images()`, which also shows the *xref* numbers of each image.

clear_with(`[value[,irect]]`)

Initialize the samples area.

Parameters

- **value** (*int*) – if specified, values from 0 to 255 are valid. Each color byte of each pixel will be set to this value, while alpha will be set to 255 (non-transparent) if present. If omitted, then all bytes (including any alpha) are cleared to `0x00`.
- **irect** (*irect_like*) – the area to be cleared. Omit to clear the whole pixmap. Can only be specified, if *value* is also specified.

tint_with(*red, green, blue*)

Colorize (tint) a pixmap with a color provided as an integer triple (red, green, blue). Only colorspaces `CS_GRAY` and `CS_RGB` are supported, others are ignored with a warning.

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

Parameters

- **red** (*int*) – *red* component.
- **green** (*int*) – *green* component.
- **blue** (*int*) – *blue* component.

gamma_with(*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 .

Parameters

n (*int*) – determines the new pixmap (samples) size. For example, a value of 2 divides width and height by 4 and thus results in a size of one 16^{th} of the original. Values less than 1 are ignored with a warning.

Note: Use this methods to reduce a pixmap’s size retaining its proportion. The pixmap is changed “in place”. If you want to keep original and also have more granular choices, use the resp. copy constructor above.

pixel(*x, y*)

New in version: 1.14.5: Return the value of the pixel at location (x, y) (column, line).

Parameters

- **x** (*int*) – the column number of the pixel. Must be in `range(pix.width)`.
- **y** (*int*) – the line number of the pixel, Must be in `range(pix.height)`.

Return type

list

Returns

a list of color values and, potentially the alpha value. Its length and content depend on the pixmap's colorspace and the presence of an alpha. For RGBA pixmaps the result would e.g. be `[r, g, b, a]`. All items are integers in `range(256)`.

set_pixel(*x*, *y*, *color*)

New in version 1.14.7: Manipulate the pixel at location (*x*, *y*) (column, line).

Parameters

- **x** (*int*) – the column number of the pixel. Must be in `range(pix.width)`.
- **y** (*int*) – the line number of the pixel. Must be in `range(pix.height)`.
- **color** (*sequence*) – the desired pixel value given as a sequence of integers in `range(256)`. The length of the sequence must equal `Pixmap.n`, which includes any alpha byte.

set_rect(*irect*, *color*)

New in version 1.14.8: Set the pixels of a rectangle to a value.

Parameters

- **irect** (*irect_like*) – the rectangle to be filled with the value. The actual area is the intersection of this parameter and `Pixmap.irect`. For an empty intersection (or an invalid parameter), no change will happen.
- **color** (*sequence*) – the desired value, given as a sequence of integers in `range(256)`. The length of the sequence must equal `Pixmap.n`, which includes any alpha byte.

Return type

bool

Returns

False if the rectangle was invalid or had an empty intersection with `Pixmap.irect`, else *True*.

Note:

1. This method is equivalent to `Pixmap.set_pixel()` executed for each pixel in the rectangle, but is obviously **very much faster** if many pixels are involved.
 2. This method can be used similar to `Pixmap.clear_with()` to initialize a pixmap with a certain color like this: `pix.set_rect(pix.irect, (255, 255, 0))` (RGB example, colors the complete pixmap with yellow).
-

set_origin(*x*, *y*)

- New in v1.17.7

Set the *x* and *y* values of the pixmap's top-left point.

Parameters

- **x** (*int*) – *x* coordinate
- **y** (*int*) – *y* coordinate

set_dpi(*xres*, *yres*)

- New in v1.16.17
- Changed in v1.18.0: When saving as a PNG image, these values will be stored now.

Set the resolution (dpi) in x and y direction.

Parameters

- **xres** (*int*) – resolution in x direction.
- **yres** (*int*) – resolution in y direction.

set_alpha(*alphavalues*, *premultiply*=1, *opaque*=None)

- Changed in v 1.18.13

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

Parameters

- **alphavalues** (*bytes*, *bytearray*, *BytesIO*) – the new alpha values. If provided, its length must be at least *width * height*. If omitted (None), all alpha values are set to 255 (no transparency). *Changed in version 1.14.13*: *io.BytesIO* is now also accepted.
- **premultiply** (*bool*) – *New in v1.18.13*: whether to premultiply color components with the alpha value.
- **opaque** (*list*, *tuple*) – ignore the alpha value and set this color to fully transparent. A sequence of integers in *range*(256) with a length of *Pixmap.n*. Default is *None*. For example, a typical choice for RGB would be *opaque*=(255, 255, 255) (white).

invert_irect([*irect*])

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

Parameters

irect (*irect_like*) – The area to be inverted. Omit to invert everything.

copy(*source*, *irect*)

Copy the *irect* part of the *source* pixmap into the corresponding area of this one. The two pixmaps may have different dimensions and can each have *CS_GRAY* or *CS_RGB* colorspace, but they currently **must** have the same alpha property². The copy mechanism automatically adjusts discrepancies between source and target like so:

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

Between *irect* and the target pixmap's rectangle, an "intersection" is calculated at first. This takes into account the rectangle coordinates and the current attribute values *Pixmap.x* and *Pixmap.y* (which you are free to modify for this purpose via *Pixmap.set_origin()*). Then the corresponding data of this intersection are copied. If the intersection is empty, nothing will happen.

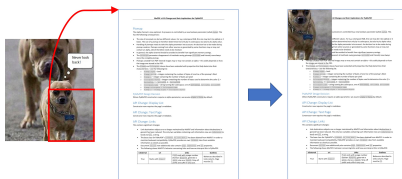
Parameters

- **source** (*Pixmap*) – source pixmap.
- **irect** (*irect_like*) – The area to be copied.

² To also set the alpha property, add an additional step to this method by dropping or adding an alpha channel to the result.

Note: Example: Suppose you have two pixmaps, `pix1` and `pix2` and you want to copy the lower right quarter of `pix2` to `pix1` such that it starts at the top-left point of `pix1`. Use the following snippet:

```
>>> # safeguard: set top-left of pix1 and pix2 to (0, 0)
>>> pix1.set_origin(0, 0)
>>> pix2.set_origin(0, 0)
>>> # compute top-left coordinates of pix2 region to copy
>>> x1 = int(pix2.width / 2)
>>> y1 = int(pix2.height / 2)
>>> # shift top-left of pix2 such, that the to-be-copied
>>> # area starts at (0, 0):
>>> pix2.set_origin(-x1, -y1)
>>> # now copy ...
>>> pix1.copy(pix2, (0, 0, x1, y1))
```



save(*filename*, *output=None*)

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 silently ignored.

Parameters

- **filename** (*str*, *Path*, *file*) – The file to save to. May be provided as a string, as a `pathlib.Path` or as a Python file object. In the latter two cases, the filename is taken from the resp. object. The filename’s extension determines the image format, which can be overruled by the output parameter.
- **output** (*str*) – The requested image format. The default is the filename’s extension. If not recognized, *png* is assumed. For other possible values see [Supported Output Image Formats](#).

pdfocr_save(*filename*, *compress=True*, *language='eng'*)

- New in v1.19.0

Perform text recognition using Tesseract and save the image as a 1-page PDF with an OCR text layer.

Parameters

- **filename** (*str*, *fp*) – identifies the file to save to. May be either a string or a pointer to a file opened with “wb” (includes `io.BytesIO()` objects).
- **compress** (*bool*) – whether to compress the resulting PDF, default is `True`.
- **language** (*str*) – the languages occurring in the image. This must be specified in Tesseract format. Default is “eng” for English. Use “+”-separated Tesseract language codes for multiple languages, like “eng+spa” for English and Spanish.

Note: Will fail if Tesseract is not installed or if the environment variable “TESSDATA_PREFIX” is not set to the tessdata folder name. This is what you would typically see on a Windows platform:

```
>>> print(os.environ["TESSDATA_PREFIX"])
C:\Program Files\Tesseract-OCR\tessdata
```

Respectively on a Linux system:

```
>>> print(os.environ["TESSDATA_PREFIX"])
/usr/share/tesseract-ocr/4.00/tessdata
```

pdfocr_tobytes(*compress=True, language='eng'*)

- New in v1.19.0

Perform text recognition using Tesseract and convert the image to a 1-page PDF with an OCR text layer. Internally invokes [Pixmap.pdfocr_save\(\)](#).

Returns

A 1-page PDF file in memory. Could be opened like `doc=fitz.open("pdf", pix.pdfocr_tobytes())`, and text extractions could be performed on its `page=doc[0]`.

Note: Another possible use is insertion into some pdf. The following snippet reads the images of a folder and stores them as pages in a new PDF that contain an OCR text layer:

```
doc = fitz.open()
for imgfile in os.listdir(folder):
    pix = fitz.Pixmap(imgfile)
    imgpdf = fitz.open("pdf", pix.pdfocr_tobytes())
    doc.insert_pdf(imgpdf)
    pix = None
    imgpdf.close()
doc.save("ocr-images.pdf")
```

tobytes(*output='png'*)

New in version 1.14.5: Return the pixmap as a *bytes* memory object of the specified format – similar to [save\(\)](#).

Parameters

output (*str*) – The requested image format. The default is “png” for which this function equals [tobytes\(\)](#). For other possible values see [Supported Output Image Formats](#).

Return type

bytes

pil_save(**args, **kwargs*)

- New in v1.17.3

Write the pixmap as an image file using Pillow. Use this method for output unsupported by MuPDF. Examples are

- Formats JPEG, JPX, J2K, WebP, etc.
- Storing EXIF information.
- If you do not provide dpi information, the values *xres*, *yres* stored with the pixmap are automatically used.

A simple example: `pix.pil_save("some.jpg", optimize=True, dpi=(150, 150))`. For details on other parameters see the Pillow documentation.

Note: (Changed in v1.18.0) `Pixmap.save()` now also sets dpi from `xres` / `yres` automatically, when saving a PNG image.

If Pillow is not installed an `ImportError` exception is raised.

`pil_tobytes(*args, **kwargs)`

- New in v1.17.3

Return an image as a bytes object in the specified format using Pillow. For example `stream = pix.pil_tobytes(format="JPEG", optimize=True)`. Also see above. For details on other parameters see the Pillow documentation. If Pillow is not installed, an `ImportError` exception is raised.

Return type

bytes

`warp(quad, width, height)`

- New in v1.19.3

Return a new pixmap by “warping” the quad such that the quad corners become the new pixmap’s corners. The target pixmap’s `irect` will be `(0, 0, width, height)`.

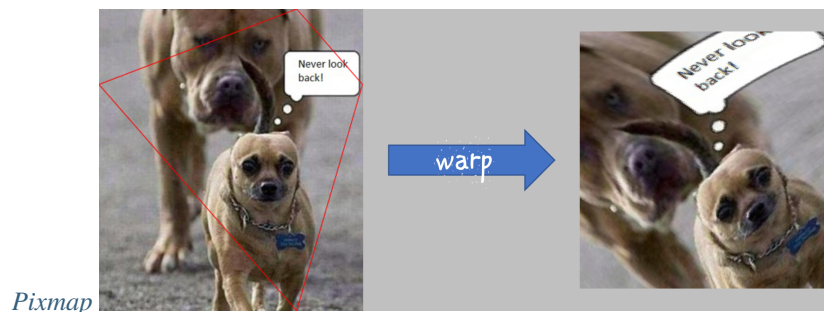
Parameters

- **`quad`** (*quad_like*) – a convex quad with coordinates inside `Pixmap.irect` (including the border points).
- **`width`** (*int*) – desired resulting width.
- **`height`** (*int*) – desired resulting height.

Returns

A new pixmap where the quad corners are mapped to the pixmap corners in a clockwise fashion: `quad.ul -> irect.tl`, `quad.ur -> irect.tr`, etc.

Return type



`color_count(colors=False, clip=None)`

- New in v1.19.2
- Changed in v1.19.3

Determine the pixmap’s unique colors and their count.

Parameters

- **colors** (*bool*) – (*changed in v1.19.3*) If **True** return a dictionary of color pixels and their usage count, else just the number of unique colors.
- **clip** (*rect_like*) – a rectangle inside *Pixmap.irect*. If provided, only those pixels are considered. This allows inspecting sub-rectangles of a given pixmap directly – instead of building sub-pixmaps.

Return type

dict or int

Returns

either the number of colors, or a dictionary with the items `pixel: count`. The pixel key is a bytes object of length *Pixmap.n*.

Note: To recover the **tuple** of a pixel, use `tuple(colors.keys()[i])` for the *i*-th item.

- The response time depends on the pixmap's samples size and may be more than a second for very large pixmaps.
 - Where applicable, pixels with different alpha values will be treated as different colors.
-

color_topusage(*clip=None*)

- New in v1.19.3

Return the most frequently used color and its relative frequency.

Parameters

clip (*rect_like*) – a rectangle inside *Pixmap.irect*. If provided, only those pixels are considered. This allows inspecting sub-rectangles of a given pixmap directly – instead of building sub-pixmaps.

Return type

tuple[float, bytes]

Returns

A tuple (`ratio`, `pixel`) where $0 < \text{ratio} \leq 1$ and *pixel* is the pixel value of the color. Use this to decide if the image is “almost” unicolor: a response (`0.95`, `b"\x00\x00\x00"`) means that 95% of all pixels are black.

alpha

Indicates whether the pixmap contains transparency information.

Type

bool

digest

The MD5 hashcode (16 bytes) of the pixmap. This is a technical value used for unique identifications.

Type

bytes

colorspace

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

Type

Colorspace

stride

Contains the length of one row of image data in *Pixmap.samples*. This is primarily used for calculation purposes. The following expressions are true:

- `len(samples) == height * stride`
- `width * n == stride`

Type

int

is_monochrome

- New in v1.19.2

Is True for a gray pixmap which only has the colors black and white.

Type

bool

is_unicolor

- New in v1.19.2

Is True if all pixels are identical (any colorspace). Where applicable, pixels with different alpha values will be treated as different colors.

Type

bool

irect

Contains the *IRect* of the pixmap.

Type*IRect***samples**

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

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

Note:

- The underlying data is typically a **large** memory area, from which a bytes copy is made for this attribute ... each time you access it: for example an RGB-rendered letter page has a samples size of almost 1.4 MB. So consider assigning a new variable to it or use the memoryview version *Pixmap.samples_mv* (new in v1.18.17).
- Any changes to the underlying data are available only after accessing this attribute again. This is different from using the memoryview version.

Type

bytes

samples_mv

- New in v1.18.17

Like *Pixmap.samples*, but in Python memoryview format. It is built pointing to the memory in the pixmap – not from a copy of it. So its creation speed is independent from the pixmap size, and any changes to pixels will be available immediately.

Copies like `bytearray(pix.samples_mv)`, or `bytes(pixmap.samples_mv)` are equivalent to and can be used in place of `pix.samples`.

We also have `len(pix.samples) == len(pix.samples_mv)`.

Look at this example from a 2 MB JPEG: the memoryview is **ten thousand times faster**:

```
In [3]: %timeit len(pix.samples_mv)
367 ns ± 1.75 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
In [4]: %timeit len(pix.samples)
3.52 ms ± 57.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Type

memoryview

samples_ptr

- New in v1.18.17

Python pointer to the pixel area. This is a special integer format, which can be used by supporting applications (such as PyQt) to directly address the samples area and thus build their images extremely fast. For example:

```
img = QtGui.QImage(pix.samples, pix.width, pix.height, format) # (1)
img = QtGui.QImage(pix.samples_ptr, pix.width, pix.height, format) # (2)
```

Both of the above lead to the same Qt image, but (2) can be **many hundred times faster**, because it avoids an additional copy of the pixel area.

Type

int

size

Contains `len(pixmap)`. This will generally equal `len(pix.samples)` plus some platform-specific value for defining other attributes of the object.

Type

int

width**w**

Width of the region in pixels.

Type

int

height

h

Height of the region in pixels.

Type

int

xX-coordinate of top-left corner in pixels. Cannot directly be changed – use [`Pixmap.set_origin\(\)`](#).**Type**

int

yY-coordinate of top-left corner in pixels. Cannot directly be changed – use [`Pixmap.set_origin\(\)`](#).**Type**

int

n

Number of components per pixel. This number depends on colorspace and alpha. If colorspace is not *None* (stencil masks), then `Pixmap.n - Pixmap.alpha == pixmap.colorsplace.n` is true. If colorspace is *None*, then `n == alpha == 1`.

Type

int

xres

Horizontal resolution in dpi (dots per inch). Please also see [*resolution*](#). Cannot directly be changed – use [`Pixmap.set_dpi\(\)`](#).

Type

int

yres

Vertical resolution in dpi (dots per inch). Please also see [*resolution*](#). Cannot directly be changed – use [`Pixmap.set_dpi\(\)`](#).

Type

int

interpolate

An information-only boolean flag set to *True* if the image will be drawn using “linear interpolation”. If *False* “nearest neighbour sampling” will be used.

Type

bool

6.15.1 Supported Input Image Formats

The following file types are supported as **input** to construct pixmaps: **BMP, JPEG, GIF, TIFF, JXR, JPX, PNG, PAM** and all of the **Portable Anymap** family (**PBM, PGM, PNM, PPM**). This support is two-fold:

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

SVG images are only supported via method 2 above, not directly as pixmaps. But remember: the result of this is a **raster image** as is always the case with pixmaps¹.

6.15.2 Supported Output Image Formats

A number of image **output** formats are supported. You have the option to either write an image directly to a file (*Pixmap.save()*), or to generate a bytes object (*Pixmap.tobytes()*). Both methods accept a 3-letter string identifying the desired format (**Format** column below). Please note that not all combinations of pixmap colorspace, transparency support (alpha) and image format are possible.

Format	Colorspaces	alpha	Extensions	Description
pam	gray, rgb, cmyk	yes	.pam	Portable Arbitrary Map
pbm	gray, rgb	no	.pbm	Portable Bitmap
pgm	gray, rgb	no	.pgm	Portable Graymap
png	gray, rgb	yes	.png	Portable Network Graphics
pnm	gray, rgb	no	.pnm	Portable Anymap
ppm	gray, rgb	no	.ppm	Portable Pixmap
ps	gray, rgb, cmyk	no	.ps	Adobe PostScript Image
psd	gray, rgb, cmyk	yes	.psd	Adobe Photoshop Document

Note:

- Not all image file types are supported (or at least common) on all OS platforms. E.g. PAM and the Portable Anymap formats are rare or even unknown on Windows.
 - Especially pertaining to CMYK colorspaces, you can always convert a CMYK pixmap to an RGB pixmap with *rgb_pix = fitz.Pixmap(fitz.csRGB, cmyk_pix)* and then save that in the desired format.
 - As can be seen, MuPDF’s image support range is different for input and output. Among those supported both ways, PNG is probably the most popular. We recommend using Pillow whenever you face a support gap.
 - We also recommend using “ppm” formats as input to tkinter’s *PhotoImage* method like this: *tkimg = tkinter.PhotoImage(data=pix.tobytes(“ppm”))* (also see the tutorial). This is **very fast (60 times faster than PNG)** and will work under Python 2 or 3.
-

6.16 Point

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

¹ If you need a **vector image** from the SVG, you must first convert it to a PDF. Try *Document.convert_to_pdf()*. If this is not good enough, look for other SVG-to-PDF conversion tools like the Python packages *svglib*, *CairoSVG*, *Uniconvertor* or the Java solution *Apache Batik*. Have a look at our Wiki for more examples.

Attribute / Method	Description
<code>Point.distance_to()</code>	calculate distance to point or rect
<code>Point.norm()</code>	the Euclidean norm
<code>Point.transform()</code>	transform point with a matrix
<code>Point.abs_unit</code>	same as unit, but positive coordinates
<code>Point.unit</code>	point coordinates divided by <i>abs(point)</i>
<code>Point.x</code>	the X-coordinate
<code>Point.y</code>	the Y-coordinate

Class API

class Point

`__init__(self)`

`__init__(self, x, y)`

`__init__(self, point)`

`__init__(self, sequence)`

Overloaded constructors.

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

With another point specified, a **new copy** will be crated, “sequence” is a Python sequence of 2 numbers (see *Using Python Sequences as Arguments in PyMuPDF*).

Parameters

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

`distance_to(x[, unit])`

Calculate the distance to *x*, which may be *point_like* or *rect_like*. The distance is given in units of either pixels (default), inches, centimeters or millimeters.

Parameters

- **x** (*point_like, rect_like*) – to which to compute the distance.
- **unit** (*str*) – the unit to be measured in. One of “px”, “in”, “cm”, “mm”.

Return type

float

Returns

the distance to *x*. If this is *rect_like*, then the distance

- is the length of the shortest line connecting to one of the rectangle sides
- is calculated to the **finite version** of it
- is zero if it **contains** the point

norm()

- New in version 1.16.0

Return the Euclidean norm (the length) of the point as a vector. Equals result of function *abs()*.

transform(*m*)

Apply a matrix to the point and replace it with the result.

Parameters

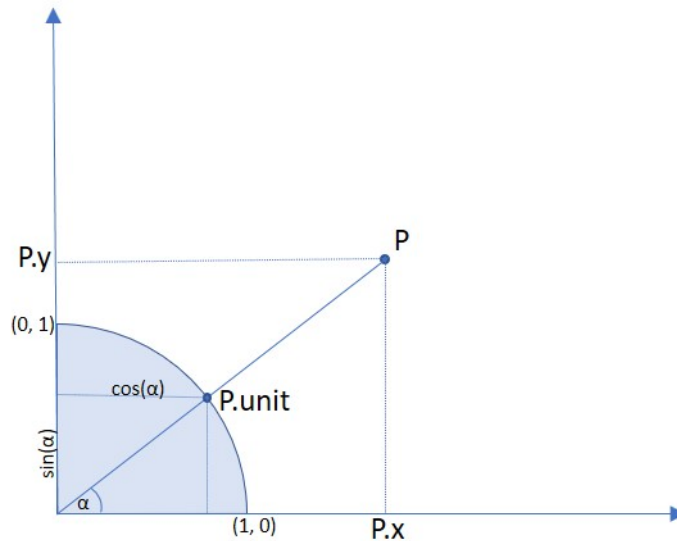
m (*matrix_like*) – The matrix to be applied.

Return type

Point

unit

Result of dividing each coordinate by *norm(point)*, the distance of the point to (0,0). This is a vector of length 1 pointing in the same direction as the point does. Its x, resp. y values are equal to the cosine, resp. sine of the angle this vector (and the point itself) has with the x axis.

**Type**

Point

abs_unit

Same as *unit* above, replacing the coordinates with their absolute values.

Type

Point

x

The x coordinate

Type

float

y

The y coordinate

Type
float

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
- Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.

6.17 Quad

Represents a four-sided mathematical shape (also called “quadrilateral” or “tetragon”) in the plane, defined as a sequence of four *Point* objects *ul*, *ur*, *ll*, *lr* (conveniently called upper left, upper right, lower left, lower right).

Quads can **be obtained** as results of text search methods (*Page.search_for()*), and they **are used** to define text marker annotations (see e.g. *Page.add_squiggly_annot()* and friends), and in several draw methods (like *Page.draw_quad()* / *Shape.draw_quad()*, *Page.draw_oval()* / *Shape.draw_quad()*).

Note:

- If the corners of a rectangle are transformed with a **rotation**, **scale** or **translation** *Matrix*, then the resulting quad is **rectangular** (= congruent to a rectangle), i.e. all of its corners again enclose angles of 90 degrees. Property *Quad.is_rectangular* checks whether a quad can be thought of being the result of such an operation.
- This is not true for all matrices: e.g. shear matrices produce parallelograms, and non-invertible matrices deliver “degenerate” tetragons like triangles or lines.
- Attribute *Quad.rect* obtains the envelopping rectangle. Vice versa, rectangles now have attributes *Rect.quad*, resp. *IRect.quad* to obtain their respective tetragon versions.

Methods / Attributes	Short Description
<i>Quad.transform()</i>	transform with a matrix
<i>Quad.morph()</i>	transform with a point and matrix
<i>Quad.ul</i>	upper left point
<i>Quad.ur</i>	upper right point
<i>Quad.ll</i>	lower left point
<i>Quad.lr</i>	lower right point
<i>Quad.is_convex</i>	true if quad is a convex set
<i>Quad.is_empty</i>	true if quad is an empty set
<i>Quad.is_rectangular</i>	true if quad is congruent to a rectangle
<i>Quad.rect</i>	smallest containing <i>Rect</i>
<i>Quad.width</i>	the longest width value
<i>Quad.height</i>	the longest height value

Class API

```
class Quad
```

```
    __init__(self)
```

`__init__(self, ul, ur, ll, lr)`

`__init__(self, quad)`

`__init__(self, sequence)`

Overloaded constructors: “ul”, “ur”, “ll”, “lr” stand for *point_like* objects (the four corners), “sequence” is a Python sequence with four *point_like* objects.

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

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

transform(*matrix*)

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

Parameters

matrix (*matrix_like*) – the matrix.

morph(*fixpoint*, *matrix*)

(*New in version 1.17.0*) “Morph” the quad with a matrix-like using a point-like as fixed point.

Parameters

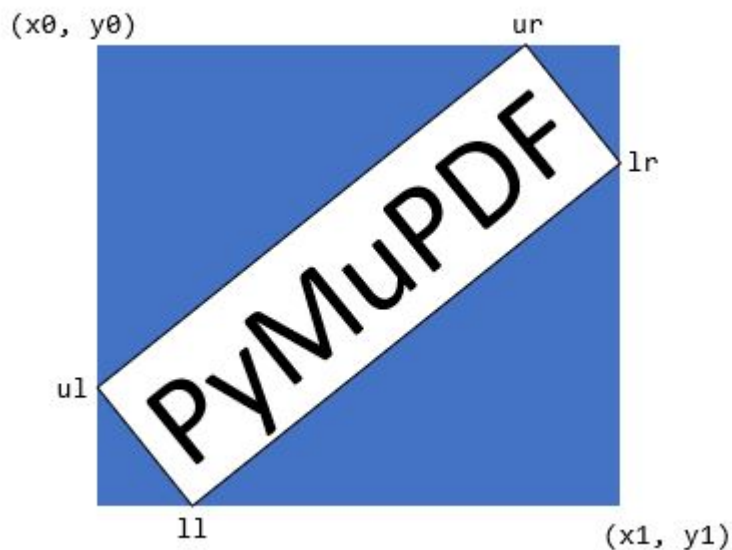
- **fixpoint** (*point_like*) – the point.
- **matrix** (*matrix_like*) – the matrix.

Returns

a new quad (no operation if this is the infinite quad).

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

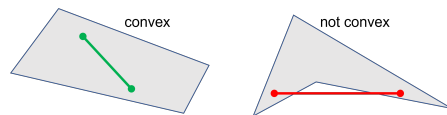
Lower right point.

Type
Point

is_convex

- New in version 1.16.1

Checks if for any two points of the quad, all points on their connecting line also belong to the quad.



Type
bool

is_empty

True if enclosed area is zero, which means that at least three of the four corners are on the same line. If this is false, the quad may still be degenerate or not look like a tetragon at all (triangles, parallelograms, trapezoids, ...).

Type
bool

is_rectangular

True if all corner angles are 90 degrees. This implies that the quad is **convex and not empty**.

Type
bool

width

The maximum length of the top and the bottom side.

Type
float

height

The maximum length of the left and the right side.

Type
float

6.17.1 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*.

6.17.2 Algebra and Containment Checks

Starting with v1.19.6, quads can be used in algebraic expressions like the other geometry object – the respective restrictions have been lifted. In particular, all the following combinations of containment checking are now possible:

```
{Point | IRect | Rect | Quad} in {IRect | Rect | Quad}
```

Please note the following interesting detail:

For a rectangle, only its top-left point belongs to it. Since v1.19.0, rectangles are defined to be “open”, such that its bottom and its right edge do not belong to it – including the respective corners. But for quads there exists no such notion like “openness”, so we have the following somewhat surprising implication:

```
>>> rect.br in rect
False
>>> # but:
>>> rect.br in rect.quad
True
```

6.18 Rect

Rect represents a rectangle defined by four floating point numbers x_0 , y_0 , x_1 , y_1 . They are treated as being coordinates of two diagonally opposite points. The first two numbers are regarded as the “top left” corner $P_{(x_0,y_0)}$ and $P_{(x_1,y_1)}$ as the “bottom right” one. However, these two properties need not coincide with their intuitive meanings – read on.

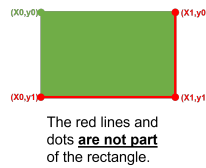
The following remarks are also valid for *IRect* objects:

- A rectangle in the sense of (Py-) MuPDF (**and PDF**) always has **borders parallel to the x- resp. y-axis**. A general orthogonal tetragon **is not a rectangle** – in contrast to the mathematical definition.
- The constructing points can be (almost! – see below) 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 four different ways:**
 1. $\text{Rect}(P_{(x_0,y_0)}, P_{(x_1,y_1)})$
 2. $\text{Rect}(P_{(x_1,y_1)}, P_{(x_0,y_0)})$
 3. $\text{Rect}(P_{(x_0,y_1)}, P_{(x_1,y_0)})$
 4. $\text{Rect}(P_{(x_1,y_0)}, P_{(x_0,y_1)})$

(**Changed in v1.19.0**) Hence some classification:

- A rectangle is called **valid** 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 **invalid**. Of the four alternatives above, **only the first** is valid. Please take into account, that in MuPDF’s coordinate system, the y-axis is oriented from **top to bottom**. Invalid rectangles have been called infinite in earlier versions.

- A rectangle is called **empty** if $x0 \geq x1$ or $y0 \geq y1$. This implies, that **invalid rectangles are also always empty**. And width (resp. height) is **set to zero** if $x0 > x1$ (resp. $y0 > y1$). In previous versions, a rectangle was empty only if one of width or height was zero.
- Rectangle coordinates **cannot be outside** the number range from `FZ_MIN_INF_RECT` = -2147483648 to `FZ_MAX_INF_RECT` = 2147483520. Both values have been chosen, because they are the smallest / largest 32bit integers that survive C float conversion roundtrips. In previous versions there was no limit for coordinate values.
- There is **exactly one “infinite” rectangle**, defined by $x0 = y0 = \text{FZ_MIN_INF_RECT}$ and $x1 = y1 = \text{FZ_MAX_INF_RECT}$. It contains every other rectangle. It is mainly used for technical purposes – e.g. when a function call should ignore a formally required rectangle argument. This rectangle is not empty.
- **Rectangles are (semi-) open:** The right and the bottom edges (including the resp. corners) are not considered part of the rectangle. This implies, that only the top-left corner ($x0$, $y0$) can ever belong to the rectangle - the other three corners never do. An empty rectangle contains no corners at all.



- Here is an overview of the changes.

Notion	Versions < 1.19.0	Versions 1.19.*
empty	$x0 = x1$ or $y0 = y1$	$x0 \geq x1$ or $y0 \geq y1$ – includes invalid rects
valid	n/a	$x0 \leq x1$ and $y0 \leq y1$
infinite	all rects where $x0 > x1$ or $y1 > y0$	exactly one infinite rect / irect!
coordinate values	all numbers	<code>FZ_MIN_INF_RECT</code> \leq number \leq <code>FZ_MAX_INF_RECT</code>
borders, corners	are parts of the rectangle	right and bottom corners and edges are outside

- There are new top level functions defining infinite and standard empty rectangles and quads, see [INFINITE_RECT\(\)](#) and friends.

Methods / Attributes	Short Description
<i>Rect.contains()</i>	checks containment of <i>point_likes</i> and <i>rect_likes</i>
<i>Rect.get_area()</i>	calculate rectangle area
<i>Rect.include_point()</i>	enlarge rectangle to also contain a point
<i>Rect.include_rect()</i>	enlarge rectangle to also contain another one
<i>Rect.intersect()</i>	common part with another rectangle
<i>Rect.intersects()</i>	checks for non-empty intersections
<i>Rect.morph()</i>	transform with a point and a matrix
<i>Rect.torect()</i>	the matrix that transforms to another rectangle
<i>Rect.norm()</i>	the Euclidean norm
<i>Rect.normalize()</i>	makes a rectangle valid
<i>Rect.round()</i>	create smallest <i>IRect</i> containing rectangle
<i>Rect.transform()</i>	transform rectangle with a matrix
<i>Rect.bottom_left</i>	bottom left point, synonym <i>bl</i>
<i>Rect.bottom_right</i>	bottom right point, synonym <i>br</i>
<i>Rect.height</i>	rectangle height
<i>Rect.irect</i>	equals result of method <i>round()</i>
<i>Rect.is_empty</i>	whether rectangle is empty
<i>Rect.is_valid</i>	whether rectangle is valid
<i>Rect.is_infinite</i>	whether rectangle is infinite
<i>Rect.top_left</i>	top left point, synonym <i>tl</i>
<i>Rect.top_right</i>	top right point, synonym <i>tr</i>
<i>Rect.quad</i>	<i>Quad</i> made from rectangle corners
<i>Rect.width</i>	rectangle width
<i>Rect.x0</i>	left corners' x coordinate
<i>Rect.x1</i>	right corners' x -coordinate
<i>Rect.y0</i>	top corners' y coordinate
<i>Rect.y1</i>	bottom corners' y coordinate

Class API

class Rect

```
__init__(self)
__init__(self, x0, y0, x1, y1)
__init__(self, top_left, bottom_right)
__init__(self, top_left, x1, y1)
__init__(self, x0, y0, bottom_right)
__init__(self, rect)
__init__(self, sequence)
```

Overloaded constructors: *top_left*, *bottom_right* stand for [*point_like*](#) objects, “sequence” is a Python sequence type of 4 numbers (see [Using Python Sequences as Arguments in PyMuPDF](#)), “rect” means another [*rect_like*](#), while the other parameters mean coordinates.

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

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

round()

Creates the smallest containing *IRect*. This is **not** the same as simply rounding the rectangle's edges: The top left corner is rounded upwards and to the left while the bottom right corner is rounded downwards and to the right.

```
>>> fitz.Rect(0.5, -0.01, 123.88, 455.123456).round()
IRect(0, -1, 124, 456)
```

1. If the rectangle is **empty**, the result is also empty.
2. **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.is_empty # rect is NOT empty
False
>>> r.round() # but its irect IS empty!
fitz.IRect(100, 100, 200, 100)
>>> r.round().is_empty
True
```

Return type*IRect***transform(*m*)**

Transforms the rectangle with a matrix and **replaces the original**. If the rectangle is empty or infinite, this is a no-operation.

Parameters

m (*Matrix*) – The matrix for the transformation.

Return type*Rect***Returns**

the smallest rectangle that contains the transformed original.

intersect(*r*)

The intersection (common rectangular area, largest rectangle contained in both) 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. If the rectangles are (mathematically) disjoint sets, then the result is invalid. If the result is valid but empty, then the rectangles touch each other in a corner or (part of) a side.

Parameters

r (*Rect*) – Second rectangle

include_rect(*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

include_point(*p*)

The smallest rectangle containing the current one and point *p* is calculated and **replaces the current** one. **The infinite rectangle remains unchanged.** To create a rectangle containing a series of points, start with (the empty) *fitz.Rect(p1, p1)* and successively include the remaining points.

Parameters

p (*Point*) – Point to include.

get_area([*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. *x in rect* and *rect.contains(x)* are equivalent.

Parameters

x (*rect_like* or *point_like*.) – the object to check.

Return type

bool

intersects(*r*)

Checks whether the rectangle and a *rect_like* “*r*” contain a common non-empty *Rect*. This will always be *False* if either is infinite or empty.

Parameters

r (*rect_like*) – the rectangle to check.

Return type

bool

to_rect(*rect*)

- New in version 1.19.3

Compute the matrix which transforms this rectangle to a given one.

Parameters

rect (*rect_like*) – the target rectangle. Must not be empty or infinite.

Return type

Matrix

Returns

a matrix *mat* such that *self * mat = rect*. Can for example be used to transform between the page and the pixmap coordinates.

Note: Suppose you want to check whether any of the words “pixmap” is invisible, because the text color equals the ambient color – e.g. white on white. We make a pixmap and check the “color environment” of each word:


```

>>> # make a pixmap of the page
>>> pix = page.get_pixmap(dpi=150)
>>> # make a matrix that transforms to pixmap coordinates
>>> mat = page.rect.torect(pix.irect)
>>> # search for text locations
>>> rlist = page.search_for("pixmap")
>>> # check color environment of each occurrence
>>> # we will check for "almost unicolor"
>>> for r in rlist:
>>>     if pix.color_topusage(clip=r * mat)[0] > 0.95:
>>>         print("'pixmap' invisible here:", r)
>>>

```

Method `Pixmap.color_topusage()` computes the percentage of pixels showing the same color.

morph(*fixpoint*, *matrix*)

- New in version 1.17.0

Return a new quad after applying a matrix to the rectangle using the fixed point *fixpoint*.

Parameters

- **fixpoint** (*point_like*) – the fixed point.
- **matrix** (*matrix_like*) – the matrix.

Returns

a new *Quad*. This a wrapper for the same-named quad method. If infinite, the infinite quad is returned.

norm()

- New in version 1.16.0

Return the Euclidean norm of the rectangle treated as a vector of four numbers.

normalize()

Replace the rectangle with its valid 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 (but may still be empty).

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 `max(x1 - x0, 0)`.

Return type

float

height

Height of the rectangle. Equals `max(y1 - y0, 0)`.

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

is_infinite

True if this is the infinite rectangle.

Type

bool

is_empty

True if rectangle is empty.

Type

bool

is_valid

True if rectangle is valid.

Type

bool

Note:

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
 - Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.
-

6.19 Shape

This class allows creating interconnected graphical elements on a PDF page. Its methods have the same meaning and name as the corresponding *Page* methods.

In fact, each *Page* draw method is just a convenience wrapper for (1) one shape draw method, (2) the *Shape.finish()* method, and (3) the *Shape.commit()* method. For page text insertion, only the *Shape.commit()* method is invoked. If many draw and text operations are executed for a page, you should always consider using a Shape object.

Several draw methods can be executed in a row and each one of them will contribute to one drawing. Once the drawing is complete, the *Shape.finish()* method must be invoked to apply color, dashing, width, morphing and other attributes.

Draw methods of this class (and *Shape.insert_textbox()*) are logging the area they are covering in a rectangle (*Shape.rect*). This property can for instance be used to set *Page.cropbox_position*.

Text insertions *Shape.insert_text()* and *Shape.insert_textbox()* implicitly execute a “finish” and therefore only require *Shape.commit()* to become effective. As a consequence, both include parameters for controlling properties like colors, etc.

Method / Attribute	Description
<code>Shape.commit()</code>	update the page's contents
<code>Shape.draw_bezier()</code>	draw a cubic Bezier curve
<code>Shape.draw_circle()</code>	draw a circle around a point
<code>Shape.draw_curve()</code>	draw a cubic Bezier using one helper point
<code>Shape.draw_line()</code>	draw a line
<code>Shape.draw_oval()</code>	draw an ellipse
<code>Shape.draw_polyline()</code>	connect a sequence of points
<code>Shape.draw_quad()</code>	draw a quadrilateral
<code>Shape.draw_rect()</code>	draw a rectangle
<code>Shape.draw_sector()</code>	draw a circular sector or piece of pie
<code>Shape.draw_squiggle()</code>	draw a squiggly line
<code>Shape.draw_zigzag()</code>	draw a zigzag line
<code>Shape.finish()</code>	finish a set of draw commands
<code>Shape.insert_text()</code>	insert text lines
<code>Shape.insert_textbox()</code>	fit text into a rectangle
<code>Shape.doc</code>	stores the page's document
<code>Shape.draw_cont</code>	draw commands since last <code>Shape.finish()</code>
<code>Shape.height</code>	stores the page's height
<code>Shape.lastPoint</code>	stores the current point
<code>Shape.page</code>	stores the owning page
<code>Shape.rect</code>	rectangle surrounding drawings
<code>Shape.text_cont</code>	accumulated text insertions
<code>Shape.totalcont</code>	accumulated string to be stored in <code>contents</code>
<code>Shape.width</code>	stores the page's width

Class API

class Shape

`__init__`(*self*, *page*)

Create a new drawing. During importing PyMuPDF, the *fitz.Page* object is being given the convenience method *new_shape()* 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.

`draw_line`(*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*.

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

Draw a squiggly (wavy, undulated) line from *point_like* objects *p1* to *p2*. An integer number of full wave periods will always be drawn, one period having a length of $4 * \textit{breadth}$. The *breadth* parameter will

be adjusted as necessary to meet this condition. The drawn line will always turn “left” when leaving $p1$ and always join $p2$ from the “right”.

Parameters

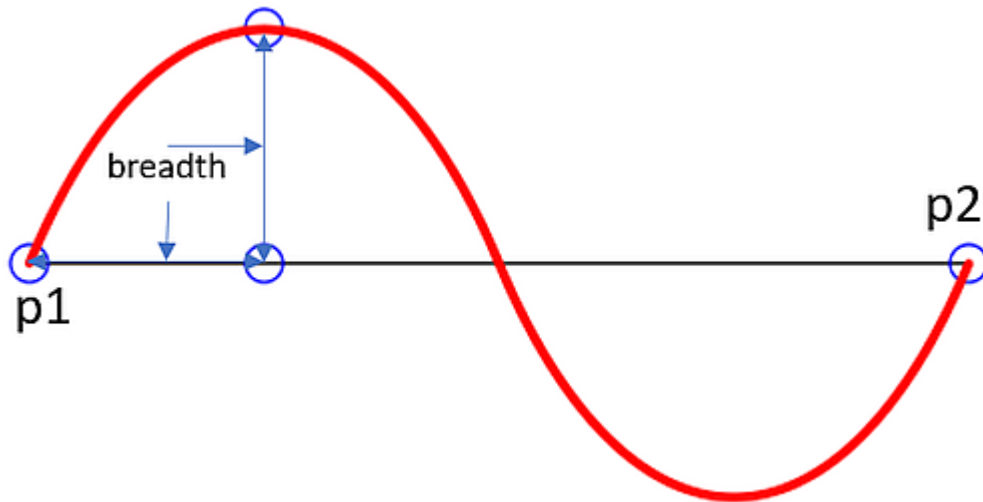
- **p1** (*point_like*) – starting point
- **p2** (*point_like*) – end point
- **breadth** (*float*) – the amplitude of each wave. The condition $2 * breadth < abs(p2 - p1)$ must be true to fit in at least one wave. See the following picture, which shows two points connected by one full period.

Return type

Point

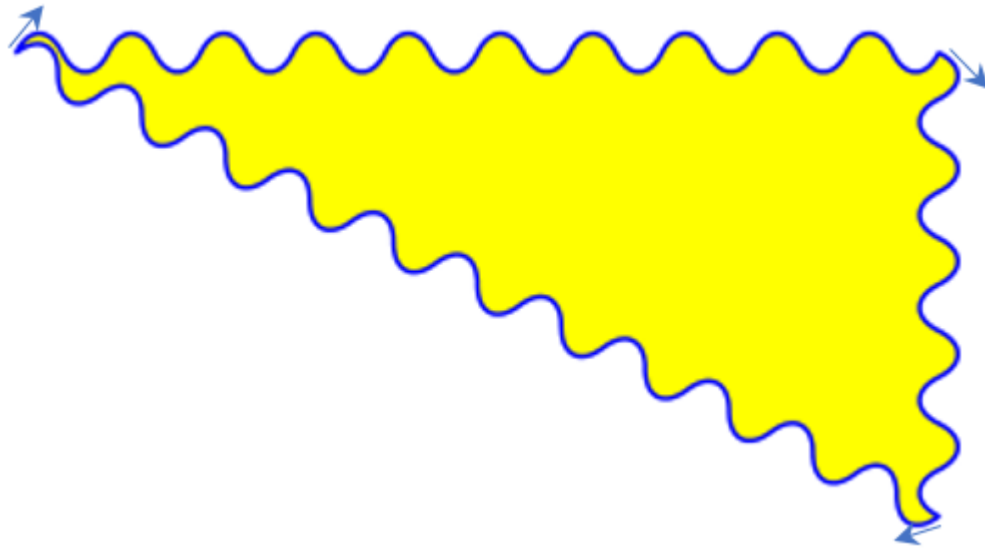
Returns

the end point, $p2$.



Here is an example of three connected lines, forming a closed, filled triangle. Little arrows indicate the stroking direction.

```
>>> import fitz
>>> doc=fitz.open()
>>> page=doc.new_page()
>>> r = fitz.Rect(100, 100, 300, 200)
>>> shape=page.new_shape()
>>> shape.draw_squiggle(r.tl, r.tr)
>>> shape.draw_squiggle(r.tr, r.br)
>>> shape.draw_squiggle(r.br, r.tl)
>>> shape.finish(color=(0, 0, 1), fill=(1, 1, 0))
>>> shape.commit()
>>> doc.save("x.pdf")
```



Note: Waves drawn are **not** trigonometric (sine / cosine). If you need that, have a look at [draw-sines.py](#).

draw_zigzag(*p1*, *p2*, *breadth*=2)

Draw a zigzag line from *point_like* objects *p1* to *p2*. Otherwise works exactly like *Shape.draw_squiggle()*.

Parameters

- **p1** (*point_like*) – starting point
- **p2** (*point_like*) – end point
- **breadth** (*float*) – the amplitude of the movement. The condition $2 * breadth < abs(p2 - p1)$ must be true to fit in at least one period.

Return type

Point

Returns

the end point, *p2*.

draw_polyline(*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 *draw_line()*).

Return type

Point

Returns

points[-1] – the last point in the argument sequence.

draw_bezier(*p1*, *p2*, *p3*, *p4*)

Draw a standard cubic Bézier curve from *p1* to *p4*, using *p2* and *p3* as control points.

All arguments are *point_like* s.

Return type

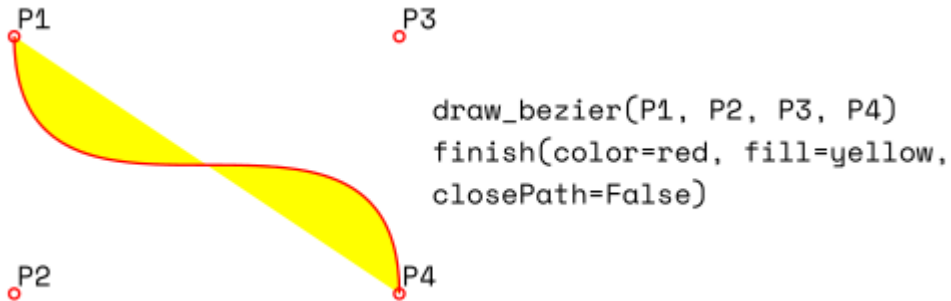
Point

Returns

the end point, *p4*.

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

Example:



draw_oval(*tetra*)

Draw an “ellipse” inside the given tetragon (quadrilateral). If it is a square, a regular circle is drawn, a general rectangle will result in an ellipse. If a quadrilateral is used instead, a plethora of shapes can be the result.

The drawing starts and ends at the middle point of the line bottom-left -> top-left corners in an anti-clockwise movement.

Parameters

tetra (*rect_like*, *quad_like*) – *rect_like* or *quad_like*.

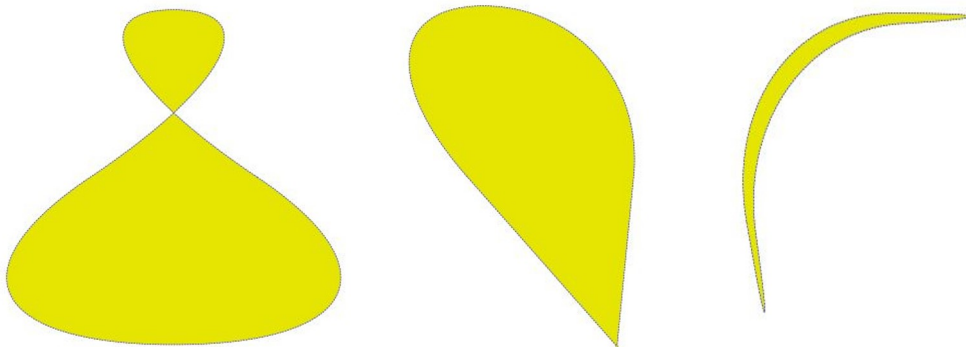
Changed in version 1.14.5: Quads are now also supported.

Return type

Point

Returns

the middle point of line *rect.bl* -> *rect.tl*, or resp. *quad.ll* -> *quad.ul*. Look at just a few examples here, or at the *quad-show?.py* scripts in the PyMuPDF-Utilities repository.



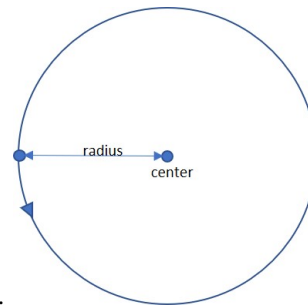
draw_circle(*center*, *radius*)

Draw a circle given its center and radius. The drawing starts and ends at point *center* - (*radius*, 0) in an **anti-clockwise** movement. This point is the middle of the enclosing square's left side.

This is a shortcut for `draw_sector(center, start, 360, fullSector=False)`. To draw the same circle in a **clockwise** movement, use -360 as degrees.

Parameters

- **center** (*point_like*) – the center of the circle.
- **radius** (*float*) – the radius of the circle. Must be positive.

Return type*Point***Returns**

`Point(center.x - radius, center.y).`

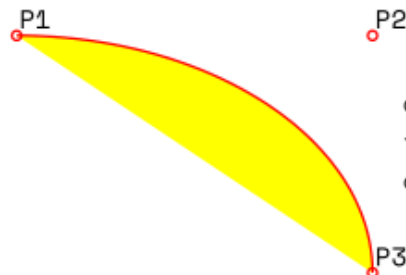
draw_curve(*p1*, *p2*, *p3*)

A special case of `draw_bezier()`: Draw a cubic Bezier curve from *p1* to *p3*. On each of the two lines *p1* -> *p2* and *p3* -> *p2* one control point is generated. Both control points will therefore be on the same side of the line *p1* -> *p3*. This guaranties that the curve's curvature does not change its sign. If the lines to *p2* intersect with an angle of 90 degrees, then the resulting curve is a quarter ellipse (resp. quarter circle, if of same length).

All arguments are *point_like*.

Return type*Point***Returns**

the end point, *p3*. The following is a filled quarter ellipse segment. The yellow area is oriented **clockwise**:



```
draw_curve(P1, P2, P3)
finish(color=red, fill=yellow,
closePath=False)
```

draw_sector(*center*, *point*, *angle*, *fullSector=True*)

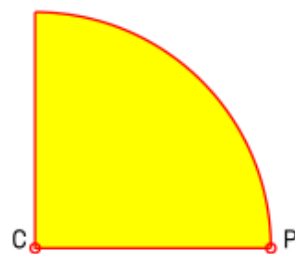
Draw a circular sector, optionally connecting the arc to the circle's center (like a piece of pie).

Parameters

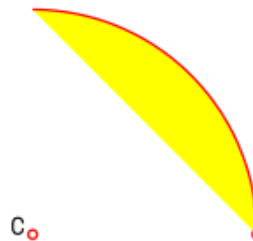
- **center** (*point_like*) – the center of the circle.
- **point** (*point_like*) – one of the two end points of the pie’s arc segment. The other one is calculated from the *angle*.
- **angle** (*float*) – the angle of the sector in degrees. Used to calculate the other end point of the arc. Depending on its sign, the arc is drawn anti-clockwise (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.

Return type*Point***Returns**

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



```
draw_sector(C, P, 90, fullSector=True)
finish(color=red, fill=yellow)
```

**draw_rect**(*rect*)

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

Parameters

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

Return type*Point***Returns**

top-left corner of the rectangle.

draw_quad(*quad*)

Draw a quadrilateral. The drawing starts and ends at the top-left corner (*Quad.ul*) in an anti-clockwise movement. It is a shortcut of *Shape.draw_polyline()* with the argument (*ul*, *ll*, *lr*, *ur*, *ul*).

Parameters

quad (*quad_like*) – where to put the tetragon on the page.

Return type*Point***Returns***Quad.ul*.

finish(*width=1*, *color=None*, *fill=None*, *lineCap=0*, *lineJoin=0*, *dashes=None*, *closePath=True*, *even_odd=False*, *morph=(fixpoint, matrix)*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

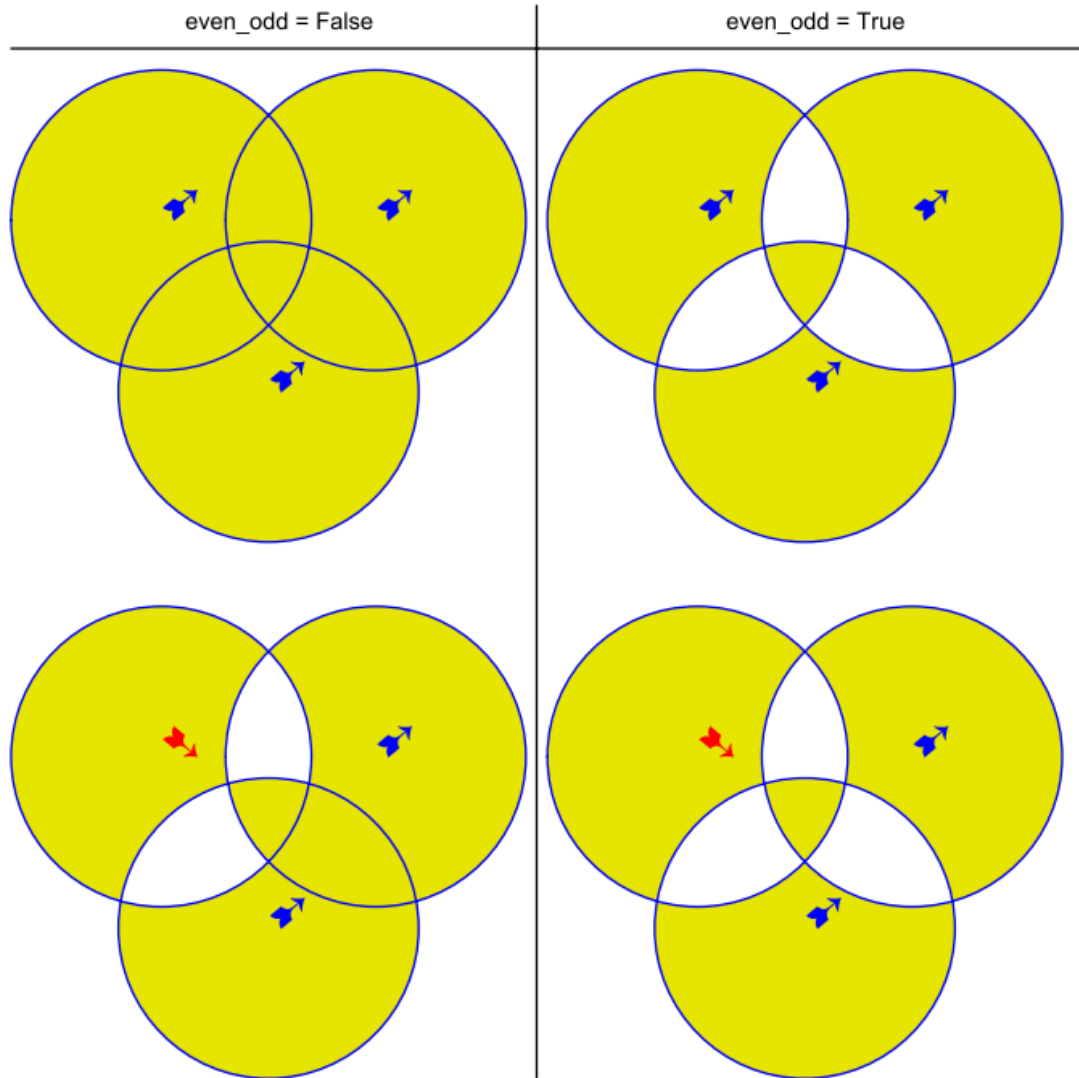
Finish a set of *draw*()* methods by applying *Common Parameters* to all of them.

It has **no effect on** *Shape.insert_text()* and *Shape.insert_textbox()*.

The method also supports **morphing the compound drawing** using *Point fixpoint* and *Matrix matrix*.

Parameters

- **morph** (*sequence*) – morph the text or the compound drawing around some arbitrary *Point* *fixpoint* by applying *Matrix* *matrix* to it. This implies that *fixpoint* is a **fixed point** of this operation: it will not change its position. Default is no morphing (*None*). The matrix can contain any values in its first 4 components, *matrix.e* == *matrix.f* == 0 must be true, however. This means that any combination of scaling, shearing, rotating, flipping, etc. is possible, but translations are not.
- **stroke_opacity** (*float*) – (*new in v1.18.1*) set transparency for stroke colors. Value < 0 or > 1 will be ignored. Default is 1 (intransparent).
- **fill_opacity** (*float*) – (*new in v1.18.1*) set transparency for fill colors. Default is 1 (intransparent).
- **even_odd** (*bool*) – request the “**even-odd rule**” for filling operations. Default is *False*, so that the “**nonzero winding number rule**” is used. These rules are alternative methods to apply the fill color where areas overlap. Only with fairly complex shapes a different behavior is to be expected with these rules. For an in-depth explanation, see *Adobe PDF References*, pp. 137 ff. Here is an example to demonstrate the difference.
- **oc** (*int*) – (*new in v1.18.4*) the *xref* number of an *OCG* or *OCMD* to make this drawing conditionally displayable.



Note: For each pixel in a shape, the following will happen:

1. Rule “**even-odd**” counts, how many areas contain the pixel. If this count is **odd**, the pixel is regarded **inside** the shape, if it is **even**, the pixel is **outside**.
2. The default rule “**nonzero winding**” in addition looks at the “*orientation*” of each area containing the pixel: it **adds 1** if an area is drawn anti-clockwise and it **subtracts 1** for clockwise areas. If the result is zero, the pixel is regarded **outside**, pixels with a non-zero count are **inside** the shape.

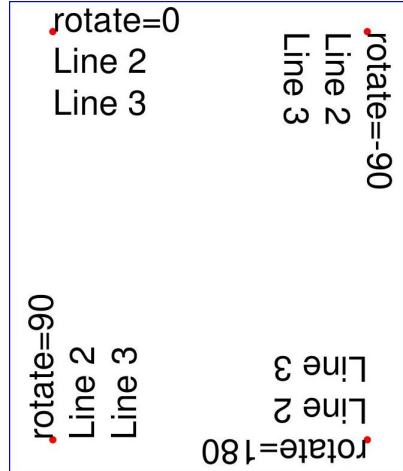
Of the four shapes in above image, the top two each show three circles drawn in standard manner (anti-clockwise, look at the arrows). The lower two shapes contain one (the top-left) circle drawn clockwise. As can be seen, area orientation is irrelevant for the right column (even-odd rule).

```
insert_text(point, text, fontsize=11, fontname='helv', fontfile=None, set_simple=False,
             encoding=TEXT_ENCODING_LATIN, color=None, lineheight=None, fill=None,
             render_mode=0, border_width=1, rotate=0, morph=None, stroke_opacity=1, fill_opacity=1,
             oc=0)
```

Insert text lines start at *point*.

Parameters

- **point** (*point_like*) – the bottom-left position of the first character of *text* in pixels. It is important to understand, how this works in conjunction with the *rotate* parameter. Please have a look at the following picture. The small red dots indicate the 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.
- **lineheight** (*float*) – a factor to override the line height calculated from font properties. If not *None*, a line height of `fontsize * lineheight` will be used.
- **stroke_opacity** (*float*) – (*new in v1.18.1*) set transparency for stroke colors. Negative values and values > 1 will be ignored. Default is 1 (intransparent).
- **fill_opacity** (*float*) – (*new in v1.18.1*) set transparency for fill colors. Default is 1 (intransparent). Use this value to control transparency of the text color. Stroke opacity **only** affects the border line of characters.
- **rotate** (*int*) – determines whether to rotate the text. Acceptable values are multiples of 90 degrees. Default is 0 (no rotation), meaning horizontal text lines oriented from left to right. 180 means text is shown upside down from **right to left**. 90 means anti-clockwise rotation, text running **upwards**. 270 (or -90) means clockwise rotation, text running **downwards**. In any case, *point* specifies the bottom-left coordinates of the first character’s rectangle. Multiple lines, if present, always follow the reading direction established by this parameter. So line 2 is located **above** line 1 in case of *rotate* = 180, etc.
- **oc** (*int*) – (*new in v1.18.4*) the *xref* number of an *OCG* or *OCMD* to make this text conditionally displayable.

Return type

int

Returns

number of lines inserted.

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

```
insert_textbox(rect, buffer, fontsize=11, fontname='helv', fontfile=None, set_simple=False,
                encoding=TEXT_ENCODING_LATIN, color=None, fill=None, render_mode=0,
                border_width=1, expandtabs=8, align=TEXT_ALIGN_LEFT, rotate=0, morph=None,
                stroke_opacity=1, fill_opacity=1, oc=0)
```

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

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](#)).
- **expandtabs** (*int*) – controls handling of tab characters *t* using the *string.expandtabs()* method **per each line**.
- **stroke_opacity** (*float*) – (*new in v1.18.1*) set transparency for stroke colors. Negative values and values > 1 will be ignored. Default is 1 (intransparent).
- **fill_opacity** (*float*) – (*new in v1.18.1*) set transparency for fill colors. Default is 1 (intransparent). Use this value to control transparency of the text color. Stroke opacity **only** affects the border line of characters.
- **rotate** (*int*) – requests text to be rotated in the rectangle. This value must be a multiple of 90 degrees. Default is 0 (no rotation). Effectively, four different values are processed: 0, 90, 180 and 270 (= -90), each causing the text to start in a different rectangle corner. Bottom-left is 90, bottom-right is 180, and -90 / 270 is top-right. See the example how text is filled in a rectangle. This argument takes precedence over morphing. See the second example, which shows text first rotated left by 90 degrees and then the whole rectangle rotated clockwise around is lower left corner.
- **oc** (*int*) – (*new in v1.18.4*) the *xref* number of an *OCG* or *OCMD* to make this text conditionally displayable.

Return type

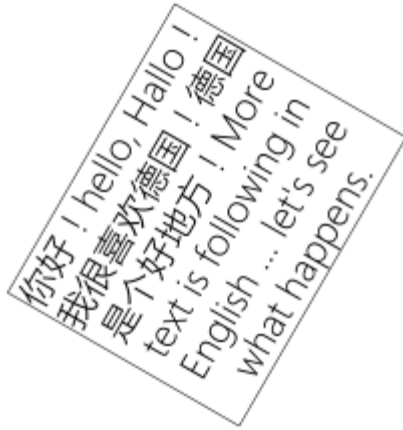
float

Returns

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

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





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

commit(*overlay=True*)

Update the page's *contents* with the accumulated drawings, followed by any text insertions. If text overlaps drawings, it will be written on top of the drawings.

Warning: Do not forget to execute this method:

If a shape is **not committed**, it will be ignored and the page will not be changed!

The method will reset attributes *Shape.rect*, *lastPoint*, *draw_cont*, *text_cont* and *totalcont*. Afterwards, the shape object can be reused for the **same page**.

Parameters

overlay (*bool*) – determine whether to put content in foreground (default) or background. Relevant only, if the page already has a non-empty *contents* object.

Attributes

doc

For reference only: the page's document.

Type

Document

page

For reference only: the owning page.

Type

Page

height

Copy of the page's height

Type

float

width

Copy of the page's width.

Type

float

draw_cont

Accumulated command buffer for **draw methods** since last finish. Every finish method will append its commands to *Shape.totalcont*.

Type

str

text_cont

Accumulated text buffer. All **text insertions** go here. This buffer will be appended to *totalcont Shape.commit()*, so that text will never be covered by drawings in the same Shape.

Type

str

rect

Rectangle surrounding drawings. This attribute is at your disposal and may be changed at any time. Its value is set to *None* when a shape is created or committed. Every *draw** method, and *Shape.insert_textbox()* update this property (i.e. **enlarge** the rectangle as needed). **Morphing** operations, however (*Shape.finish()*, *Shape.insert_textbox()*) are ignored.

A typical use of this attribute would be setting *Page.cropbox_position* to this value, when you are creating shapes for later or external use. If you have not manipulated the attribute yourself, it should reflect a rectangle that contains all drawings so far.

If you have used morphing and need a rectangle containing the morphed objects, use the following code:

```
>>> # assuming ...
>>> morph = (point, matrix)
>>> # ... recalculate the shape rectangle like so:
>>> shape.rect = (shape.rect - fitz.Rect(point, point)) * ~matrix + fitz.
↳ Rect(point, point)
```

Type*Rect***totalcont**

Total accumulated command buffer for draws and text insertions. This will be used by *Shape.commit()*.

Type

str

lastPoint

For reference only: the current point of the drawing path. It is *None* at *Shape* creation and after each *finish()* and *commit()*.

Type*Point*

6.19.1 Usage

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

```
>>> shape = page.new_shape()
>>> shape.draw1(...)
>>> shape.draw2(...)
>>> ...
>>> shape.finish(width=..., color=..., fill=..., morph=...)
>>> shape.draw3(...)
>>> shape.draw4(...)
>>> ...
>>> shape.finish(width=..., color=..., fill=..., morph=...)
>>> ...
>>> shape.insert_text*
>>> ...
>>> shape.commit()
>>> ....
```

Note:

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

6.19.2 Examples

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

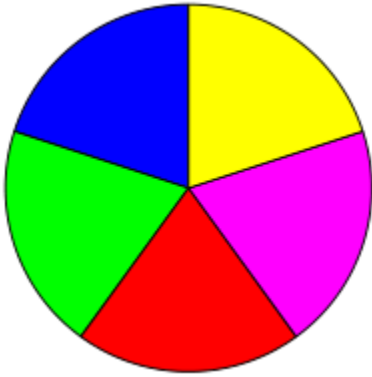
```
shape = page.new_shape() # start a new shape
cols = (...) # a sequence of RGB color triples
pieces = len(cols) # number of pieces to draw
beta = 360. / pieces # angle of each piece of pie
center = fitz.Point(...) # center of the pie
p0 = fitz.Point(...) # starting point
for i in range(pieces):
 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

(continues on next page)

(continued from previous page)

```
# now fill it but do not connect ends of the arc
shape.finish(fill=cols[i], closePath=False)
shape.commit() # update the page
```

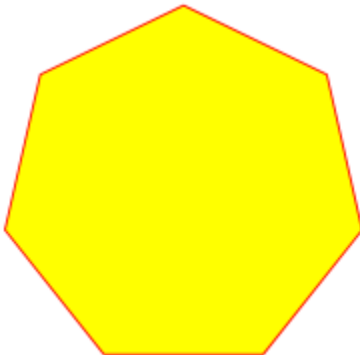
Here is an example for 5 colors:



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

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

Here is the polygon for $n = 7$:



6.19.3 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 *Page.get_fonts()* like so:

Suppose the font list has the item `[1024, 0, 'Type1', 'NimbusMonL-Bold', 'R366']`, then specify *fontname* = “/R366”, *fontfile* = *None* to use font *NimbusMonL-Bold*.

fontfile (*str*)

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

set_simple (*bool*)

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

fontsize (*float*)

Font size of text.

dashes (*str*)

Causes lines to be drawn dashed. The general format is “[*n m*] *p*” of (up to) 3 floats denoting pixel lengths. *n* is the dash length, *m* (optional) is the subsequent gap length, and *p* (the “phase” - **required**, even if 0!) specifies how many pixels should be skipped before the dashing starts. If *m* is omitted, it defaults to *n*.

A continuous line (no dashes) is drawn with “[] 0” or *None* or “”. Examples:

- Specifying “[3 4] 0” means dashes of 3 and gaps of 4 pixels following each other.
- “[3 3] 0” and “[3] 0” do the same thing.

For (the rather complex) details on how to achieve sophisticated dashing effects, see *Adobe PDF References*, page 217.

color / fill (*list, tuple*)

Stroke and fill colors can be specified as tuples or list of floats from 0 to 1. These sequences must have a length of 1 (GRAY), 3 (RGB) or 4 (CMYK). For GRAY colorspace, a single float instead of the unwieldy (*float*,) or [*float*] is also accepted. Accept (default) or use `None` to not use the parameter.

To simplify color specification, method `getColor()` in `fitz.utils` may be used to get predefined RGB color triples by name. It accepts a string as the name of the color and returns the corresponding triple. The method knows over 540 color names – see section [Color Database](#).

Please note that the term *color* usually means “stroke” color when used in conjunction with fill color.

If letting default a color parameter to `None`, then no resp. color selection command will be generated. If *fill* and *color* are both `None`, then the drawing will contain no color specification. But it will still be “stroked”, which causes PDF’s default color “black” be used by Adobe Acrobat and all other viewers.

stroke_opacity / fill_opacity (*floats*)

Both values are floats in range [0, 1]. Negative values or values > 1 will ignored (in most cases). Both set the transparency such that a value 0.5 corresponds to 50% transparency, 0 means invisible and 1 means intranparent. For e.g. a rectangle the stroke opacity applies to its border and fill opacity to its interior.

For text insertions (`Shape.insert_text()` and `Shape.insert_textbox()`), use *fill_opacity* for the text. At first sight this seems surprising, but it becomes obvious when you look further down to *render_mode*: *fill_opacity* applies to the yellow and *stroke_opacity* applies to the blue color.

border_width (*float*)

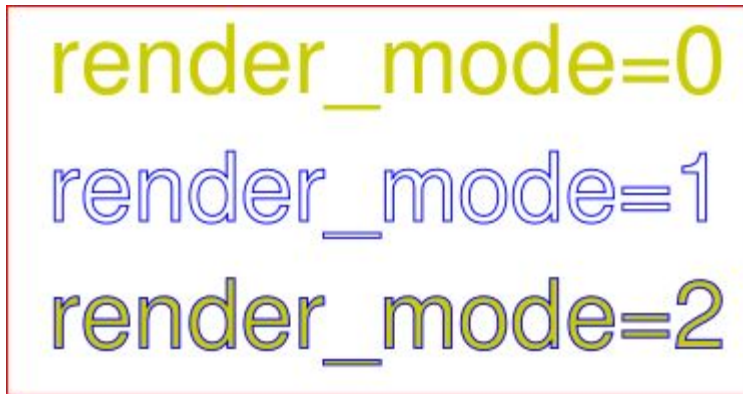
Set the border width for text insertions. New in v1.14.9. Relevant only if the render mode argument is used with a value greater zero.

render_mode (*int*)

New in version 1.14.9: Integer in range(8) which controls the text appearance (`Shape.insert_text()` and `Shape.insert_textbox()`). See page 246 in *Adobe PDF References*. New in v1.14.9. These methods now also differentiate between fill and stroke colors.

- For default 0, only the text fill color is used to paint the text. For backward compatibility, using the *color* parameter instead also works.
- For render mode 1, only the border of each glyph (i.e. text character) is drawn with a thickness as set in argument *border_width*. The color chosen in the *color* argument is taken for this, the *fill* parameter is ignored.
- For render mode 2, the glyphs are filled and stroked, using both color parameters and the specified border width. You can use this value to simulate **bold text** without using another font: choose the same value for *fill* and *color* and an appropriate value for *border_width*.
- For render mode 3, the glyphs are neither stroked nor filled: the text becomes invisible.

The following examples use `border_width=0.3`, together with a `fontsize` of 15. Stroke color is blue and fill color is some yellow.



overlay (*bool*)

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

morph (*sequence*)

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

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

lineCap (deprecated: “roundCap”) (*int*)

Controls the look of line ends. The default value 0 lets each line end at exactly the given coordinate in a sharp edge. A value of 1 adds a semi-circle to the ends, whose center is the end point and whose diameter is the line width. Value 2 adds a semi-square with an edge length of line width and a center of the line end.

Changed in version 1.14.15

lineJoin (*int*)

New in version 1.14.15: Controls the way how line connections look like. This may be either as a sharp edge (0), a rounded join (1), or a cut-off edge (2, “butt”).

closePath (*bool*)

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

6.20 Story

- New in v1.21.0

Method / Attribute	Short Description
<code>Story.reset()</code>	“rewind” story output to its beginning
<code>Story.place()</code>	compute story content to fit in provided rectangle
<code>Story.draw()</code>	write the computed content to current page
<code>Story.element_positions()</code>	callback function logging currently processed story content
<code>Story.body</code>	the story’s underlying <i>body</i>
<code>Story.write()</code>	places and draws Story to a DocumentWriter
<code>Story.write_stabilized()</code>	iterative layout of html content to a DocumentWriter
<code>Story.write_with_links()</code>	like <code>write()</code> but also creates PDF links
<code>Story.write_stabilized_with_links()</code>	like <code>write_stabilized()</code> but also creates PDF links

Class API

class Story

`__init__(self, html=None, user_css=None, em=12, archive=None)`

Create a **story**, optionally providing HTML and CSS source. The HTML is parsed, and held within the Story as a DOM (Document Object Model).

This structure may be modified: content (text, images) may be added, copied, modified or removed by using methods of the *Xml* class.

When finished, the **story** can be written to any device; in typical usage the device may be provided by a *DocumentWriter* to make new pages.

Here are some general remarks:

- The *Story* constructor parses and validates the provided HTML to create the DOM.
- PyMuPDF provides a number of ways to manipulate the HTML source by providing access to the *nodes* of the underlying DOM. Documents can be completely built from ground up programmatically, or the existing DOM can be modified pretty arbitrarily. For details of this interface, please see the *Xml* class.
- If no (or no more) changes to the DOM are required, the story is ready to be laid out and to be fed to a series of devices (typically devices provided by a *DocumentWriter* to produce new pages).
- The next step is to place the story and write it out. This can either be done directly, by looping around calling `place()` and `draw()`, or alternatively, the looping can be handled for you using the `write()` or `write_stabilized()` methods. Which method you choose is largely a matter of taste.
 - To work in the first of these styles, the following loop should be used:
 1. Obtain a suitable device to write to; typically by requesting a new, empty page from a *DocumentWriter*.
 2. Determine one or more rectangles on the page, that should receive **story** data. Note that not every page needs to have the same set of rectangles.
 3. Pass each rectangle to the **story** to place it, learning what part of that rectangle has been filled, and whether there is more story data that did not fit. This step can be repeated several times with adjusted rectangles until the caller is happy with the results.
 4. Optionally, at this point, we can request details of where interesting items have been placed, by calling the `element_positions()` method. Items are deemed to be interesting

if their integer `heading` attribute is a non-zero (corresponding to HTML tags `h1` - `h6`), if their `id` attribute is not `None` (corresponding to HTML tag `id`), or if their `href` attribute is not `None` (responding to HTML tag `href`). This can conveniently be used for automatic generation of a Table of Contents, an index of images or the like.

5. Next, draw that rectangle out to the device with the `draw()` method.
 6. If the most recent call to `place()` indicated that all the story data had fitted, stop now.
 7. Otherwise, we can loop back. If there are more rectangles to be placed on the current device (page), we jump back to step 3 - if not, we jump back to step 1 to get a new device.
- Alternatively, in the case where you are using a `DocumentWriter`, the `write()` or `write_stabilized()` methods can be used. These handle all the looping for you, in exchange for being provided with callbacks that control the behaviour (notably a callback that enumerates the rectangles/pages to use).
- Which part of the **story** will land on which rectangle / which page, is fully under control of the `Story` object and cannot be predicted.
 - Images may be part of a **story**. They will be placed together with any surrounding text.
 - Multiple stories may - independently from each other - write to the same page. For example, one may have separate stories for page header, page footer, regular text, comment boxes, etc.

Parameters

- **html** (*str*) – HTML source code. If omitted, a basic minimum is generated (see below).
- **user_css** (*str*) – CSS source code. If provided, must contain valid CSS specifications.
- **em** (*float*) – the default text font size.
- **archive** – an `Archive` from which to load resources for rendering. Currently supported resource types are images and text fonts. If omitted, the story will not try to look up any such data and may thus produce incomplete output.

Note: Instead of an actual archive, valid arguments for **creating** an `Archive` can also be provided – in which case an archive will temporarily be constructed. So, instead of `story = fitz.Story(archive=fitz.Archive("myfolder"))`, one can also shorter write `story = fitz.Story(archive="myfolder")`.

`place(`*where*`)`

Calculate that part of the story's content, that will fit in the provided rectangle. The method maintains a pointer which part of the story's content has already been written and upon the next invocation resumes from that pointer's position.

Parameters

where (*rect_like*) – layout the current part of the content to fit into this rectangle. This must be a sub-rectangle of the page's `MediaBox`.

Return type

tuple[bool, rect_like]

Returns

a bool (int) *more* and a rectangle *filled*. If *more* == 0, all content of the story has been

written, otherwise more is waiting to be written to subsequent rectangles / pages. Rectangle *filled* is the part of *where* that has actually been filled.

draw(*dev*, *matrix=None*)

Write the content part prepared by *Story.place()* to the page.

Parameters

- **dev** – the *Device* created by *dev = writer.begin_page(mediabox)*. The device knows how to call all MuPDF functions needed to write the content.
- **matrix** (*matrix_like*) – a matrix for transforming content when writing to the page. An example may be writing rotated text. The default means no transformation (i.e. the *Identity* matrix).

element_positions(*function*, *args=None*)

Let the Story provide positioning information about certain HTML elements once their place on the current page has been computed - i.e. invoke this method **directly after** *Story.place()*.

Parameters

- **function** – a Python function taking a *ElementPosition* instance, which will be invoked by this method to process positioning information.
- **args** (*dict*) – an optional dictionary with any **additional** information that should be added to the *ElementPosition* instance passed to **function**. Like for example the current output page number. Every key in this dictionary must be a string that conforms to the rules for a valid Python identifier. The complete set of information is explained below.

reset()

Rewind the story's document to the beginning for starting over its output.

body

The *body* part of the story's DOM. Even if *html=None* has been used at story creation, the following minimum HTML source will always be available:

```
<html>
  <head></head>
  <body></body>
</html>
```

This attribute contains the *Xml* node of *body*. All relevant content for PDF production is contained between “<body>” and “</body>”.

write(*writer*, *rectfn*, *positionfn=None*, *pagefn=None*)

Places and draws Story to a *DocumentWriter*. Avoids the need for calling code to implement a loop that calls *Story.place()* and *Story.draw()* etc, at the expense of having to provide at least the *rectfn()* callback.

Parameters

- **writer** – a *DocumentWriter* or *None*.
- **rectfn** – a callable taking (*rect_num*: *int*, *filled*: *Rect*) and returning (*mediabox*, *rect*, *ctm*):

mediabox:

None or *rect* for new page.

rect:

The next *rect* into which content should be placed.

ctm:

None or a *Matrix*.

- **positionfn** – None, or a callable taking (*position: ElementPosition*): *position*:

An *ElementPosition* with an extra *.page_num* member.

Typically called multiple times as we generate elements that are headings or have an id.

- **pagefn** – None, or a callable taking (*page_num, mediabox, dev, after*); called at start (*after=0*) and end (*after=1*) of each page.

static write_stabilized(*writer, contentfn, rectfn, user_css=None, em=12, positionfn=None, pagefn=None, archive=None, add_header_ids=True*)

Static method that does iterative layout of html content to a *DocumentWriter*.

For example this allows one to add a table of contents section while ensuring that page numbers are patched up until stable.

Repeatedly creates a new *Story* from (*contentfn()*, *user_css, em, archive*) and lays it out with internal call to *Story.write()*; uses a None writer and extracts the list of *ElementPosition*'s which is passed to the next call of *contentfn()*.

When the html from *contentfn()* becomes unchanged, we do a final iteration using *writer*.

Parameters

- **writer** – A *DocumentWriter*.
- **contentfn** – A function taking a list of *ElementPositions* and returning a string containing html. The returned html can depend on the list of positions, for example with a table of contents near the start.
- **rectfn** – A callable taking (*rect_num: int, filled: Rect*) and returning (*mediabox, rect, ctm*):

mediabox:

None or rect for new page.

rect:

The next rect into which content should be placed.

ctm:

A *Matrix*.

- **pagefn** – None, or a callable taking (*page_num, medibox, dev, after*); called at start (*after=0*) and end (*after=1*) of each page.
- **archive** – .
- **add_header_ids** – If true, we add unique ids to all header tags that don't already have an id. This can help automatic generation of tables of contents.

Returns:

None.

write_with_links(*rectfn, positionfn=None, pagefn=None*)

Similar to *write()* except that we don't have a *writer* arg and we return a PDF *Document* in which links have been created for each internal html link.


```
static write_stabilized_with_links(contentfn, rectfn, user_css=None, em=12, positionfn=None,
                                  pagefn=None, archive=None, add_header_ids=True)
```

Similar to `write_stabilized()` except that we don't have a `writer` arg and instead return a PDF *Document* in which links have been created for each internal html link.

6.20.1 Element Positioning CallBack function

The callback function can be used to log information about story output. The function's access to the information is read-only: it has no way to influence the story's output.

A typical loop for executing a story with using this method would look like this:

```
HTML = """
<html>
  <head></head>
  <body>
    <h1>Header level 1</h1>
    <h2>Header level 2</h2>
    <p>Hello MuPDF!</p>
  </body>
</html>
"""

MEDIABOX = fitz.paper_rect("letter") # size of a page
WHERE = MEDIABOX + (36, 36, -36, -36) # leave borders of 0.5 inches
story = fitz.Story(html=HTML) # make the story
writer = fitz.DocumentWriter("test.pdf") # make the writer
pno = 0 # current page number
more = 1 # will be set to 0 when done
while more: # loop until all story content is processed
    dev = writer.begin_page(MEDIABOX) # make a device to write on the page
    more, filled = story.place(WHERE) # compute content positions on page
    story.element_positions(recorder, {"page": pno}) # provide page number in addition
    story.draw(dev)
    writer.end_page()
    pno += 1 # increase page number
writer.close() # close output file

def recorder(elpos):
    pass
```

6.20.1.1 Attributes of the ElementPosition class

The parameter passed to the `recorder` function is an object with the following attributes:

- `elpos.depth` (int) – depth of this element in the box structure.
- `elpos.heading` (int) – the header level, 0 if no header, 1-6 for *h1* - *h6*.
- `elpos.href` (str) – value of the `href` attribute, or None if not defined.
- `elpos.id` (str) – value of the `id` attribute, or None if not defined.
- `elpos.rect` (tuple) – element position on page.
- `elpos.text` (str) – immediate text of the element.

- `elpos.open_close` (int bit field) – bit 0 set: opens element, bit 1 set: closes element. Relevant for elements that may contain other elements and thus may not immediately be closed after being created / opened.
- `elpos.rect_num` (int) – count of rectangles filled by the story so far.
- `elpos.page_num` (int) – page number; only present when using *fitz.Story.write*()* functions.

6.21 TextPage

This class represents text and images shown on a document page. All MuPDF document types are supported.

The usual ways to create a textpage are *DisplayList.get_textpage()* and *Page.get_textpage()*. Because there is a limited set of methods in this class, there exist wrappers in *Page* which are handier to use. The last column of this table shows these corresponding *Page* methods.

For a description of what this class is all about, see Appendix 2.

Method	Description	page get_text or search method
<i>extractText()</i>	extract plain text	“text”
<i>extractTEXT()</i>	synonym of previous	“text”
<i>extractBLOCKS()</i>	plain text grouped in blocks	“blocks”
<i>extractWORDS()</i>	all words with their bbox	“words”
<i>extractHTML()</i>	page content in HTML format	“html”
<i>extractXHTML()</i>	page content in XHTML format	“xhtml”
<i>extractXML()</i>	page text in XML format	“xml”
<i>extractDICT()</i>	page content in <i>dict</i> format	“dict”
<i>extractJSON()</i>	page content in JSON format	“json”
<i>extractRAW_DICT()</i>	page content in <i>dict</i> format	“rawdict”
<i>extractRAWJSON()</i>	page content in JSON format	“rawjson”
<i>search()</i>	Search for a string in the page	<i>Page.search_for()</i>

Class API

class TextPage

`extractText()`

`extractTEXT()`

Return a string of the page’s complete text. The text is UTF-8 unicode and in the same sequence as specified at the time of document creation.

Return type

str

`extractBLOCKS()`

Textpage content as a list of text lines grouped by block. Each list items looks like this:

<code>(x0, y0, x1, y1, "lines in the block", block_no, block_type)</code>

The first four entries are the block’s bbox coordinates, *block_type* is 1 for an image block, 0 for text. *block_no* is the block sequence number. Multiple text lines are joined via line breaks.

For an image block, its bbox and a text line with some image meta information is included – **not the image content**.

This is a high-speed method with just enough information to output plain text in desired reading sequence.

Return type

list

extractWORDS()

Textpage content as a list of single words with bbox information. An item of this list looks like this:

```
(x0, y0, x1, y1, "word", block_no, line_no, word_no)
```

Everything delimited by spaces is treated as a “word”. This is a high-speed method which e.g. allows extracting text from within given areas or recovering the text reading sequence.

Return type

list

extractHTML()

Textpage content as a string in HTML format. This version contains complete formatting and positioning information. Images are included (encoded as base64 strings). You need an HTML package to interpret the output in Python. Your internet browser should be able to adequately display this information, but see *Controlling Quality of HTML Output*.

Return type

str

extractDICT()

Textpage content as a Python dictionary. Provides same information detail as HTML. See below for the structure.

Return type

dict

extractJSON()

Textpage content as a JSON string. Created by `json.dumps(TextPage.extractDICT())`. It is included for backlevel compatibility. You will probably use this method ever only for outputting the result to some file. The method detects binary image data and converts them to base64 encoded strings.

Return type

str

extractXHTML()

Textpage content as a string in XHTML format. Text information detail is comparable with [extractTEXT\(\)](#), but also contains images (base64 encoded). This method makes no attempt to re-create the original visual appearance.

Return type

str

extractXML()

Textpage content as a string in XML format. This contains complete formatting information about every single character on the page: font, size, line, paragraph, location, color, etc. Contains no images. You need an XML package to interpret the output in Python.

Return type

str

extractRAWDICT()

Textpage content as a Python dictionary – technically similar to [extractDICT\(\)](#), and it contains that information as a subset (including any images). It provides additional detail down to each character, which makes using XML obsolete in many cases. See below for the structure.

Return type

dict

extractRAWJSON()

Textpage content as a JSON string. Created by `json.dumps(TextPage.extractRAWDICT())`. You will probably use this method ever only for outputting the result to some file. The method detects binary image data and converts them to base64 encoded strings.

Return type

str

search(*needle*, *quads*=False)

- Changed in v1.18.2

Search for *string* and return a list of found locations.

Parameters

- **needle** (*str*) – the string to search for. Upper and lower cases will all match if needle consists of ASCII letters only – it does not yet work for “Ä” versus “ä”, etc.
- **quads** (*bool*) – return quadrilaterals instead of rectangles.

Return type

list

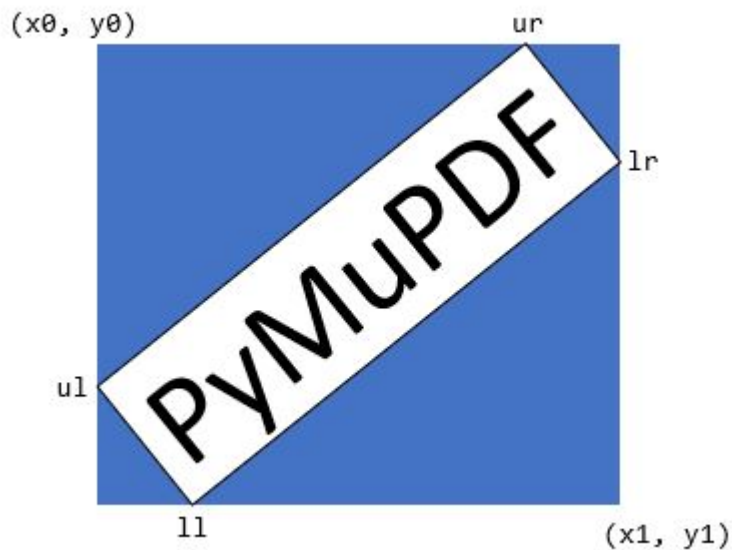
Returns

a list of [Rect](#) or [Quad](#) objects, each surrounding a found *needle* occurrence. As the search string may contain spaces, its parts may be found on different lines. In this case, more than one rectangle (resp. quadrilateral) are returned. **(Changed in v1.18.2)** The method **now supports dehyphenation**, so it will find e.g. “method”, even if it was hyphenated in two parts “meth-” and “od” across two lines. The two returned rectangles will contain “meth” (no hyphen) and “od”.

Note: Overview of changes in v1.18.2:

1. The `hit_max` parameter has been removed: all hits are always returned.
 2. The `rect` parameter of the [TextPage](#) is now respected: only text inside this area is examined. Only characters with fully contained bboxes are considered. The wrapper method [Page.search_for\(\)](#) correspondingly supports a `clip` parameter.
 3. **Hyphenated words** are now found.
 4. **Overlapping rectangles** in the same line are now automatically joined. We assume that such separations are an artifact created by multiple marked content groups, containing parts of the same search needle.
-

Example Quad versus Rect: when searching for needle “pymupdf”, then the corresponding entry will either be the blue rectangle, or, if *quads* was specified, the quad `Quad(ul, ur, ll, lr)`.

**rect**

The rectangle associated with the text page. This either equals the rectangle of the creating page or the clip parameter of `Page.get_textpage()` and text extraction / searching methods.

Note: The output of text searching and most text extractions **is restricted to this rectangle**. (X)HTML and XML output will however always extract the full page.

6.21.1 Structure of Dictionary Outputs

Methods `TextPage.extractDICT()`, `TextPage.extractJSON()`, `TextPage.extractRAWDICT()`, and `TextPage.extractRAWJSON()` return dictionaries, containing the page’s text and image content. The dictionary structures of all four methods are almost equal. They strive to map the text page’s information hierarchy of blocks, lines, spans and characters as precisely as possible, by representing each of these by its own sub-dictionary:

- A **page** consists of a list of **block dictionaries**.
- A (text) **block** consists of a list of **line dictionaries**.
- A **line** consists of a list of **span dictionaries**.
- A **span** either consists of the text itself or, for the RAW variants, a list of **character dictionaries**.
- RAW variants: a **character** is a dictionary of its origin, bbox and unicode.

All PyMuPDF geometry objects herein (points, rectangles, matrices) are represented by there “**like**” formats: a *rect_like tuple* is used instead of a *Rect*, etc. The reasons for this are performance and memory considerations:

- This code is written in C, where Python tuples can easily be generated. The geometry objects on the other hand are defined in Python source only. A conversion of each Python tuple into its corresponding geometry object would add significant – and largely unnecessary – execution time.
- A 4-tuple needs about 168 bytes, the corresponding *Rect* 472 bytes - almost three times the size. A “dict” dictionary for a text-heavy page contains 300+ bbox objects – which thus require about 50 KB storage as 4-tuples versus 140 KB as *Rect* objects. A “rawdict” output for such a page will however contain **4 to 5 thousand** bboxes, so in this case we talk about 750 KB versus 2 MB.

Please also note, that only **bboxes** (= *rect_like* 4-tuples) are returned, whereas a *TextPage* actually has the **full position information** – in *Quad* format. The reason for this decision is again a memory consideration: a *quad_like* needs 488 bytes (3 times the size of a *rect_like*). Given the mentioned amounts of generated bboxes, returning *quad_like* information would have a significant impact.

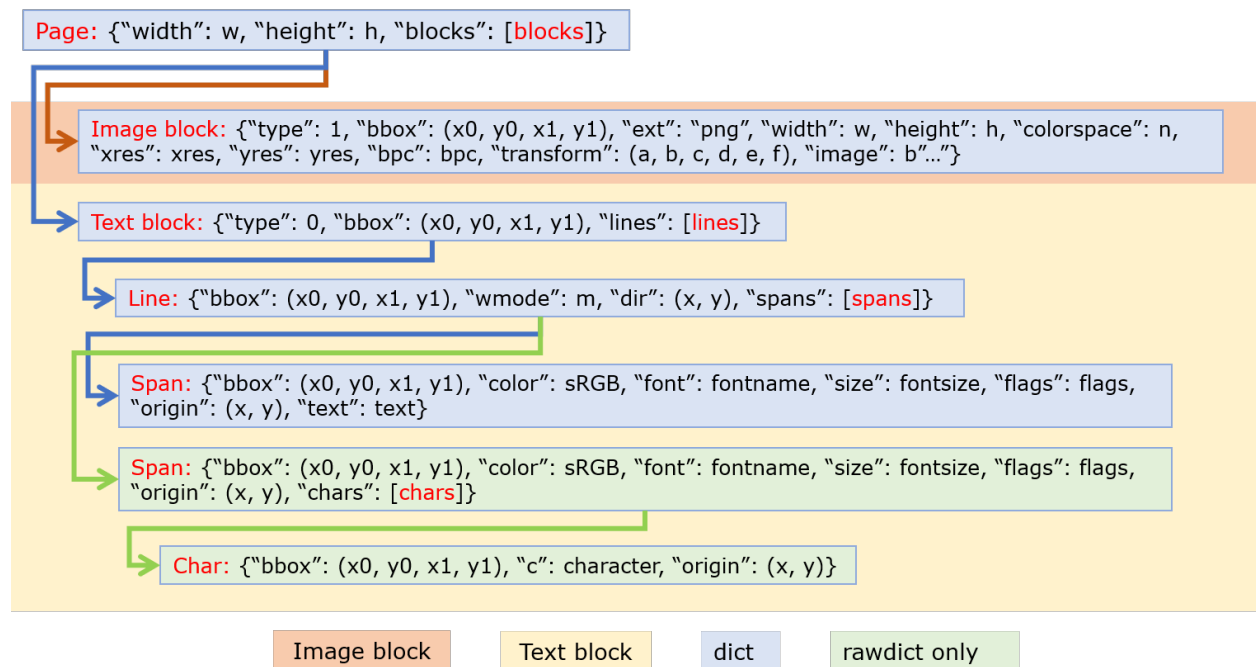
In the vast majority of cases, we are dealing with **horizontal text only**, where bboxes provide entirely sufficient information.

In addition, **the full quad information is not lost**: it can be recovered as needed for lines, spans, and characters by using the appropriate function from the following list:

- *recover_quad()* – the quad of a complete span
- *recover_span_quad()* – the quad of a character subset of a span
- *recover_line_quad()* – the quad of a line
- *recover_char_quad()* – the quad of a character

As mentioned, using these functions is ever only needed, if the text is **not written horizontally** – `line["dir"] != (1, 0)` – and you need the quad for text marker annotations (*Page.add_highlight_annot()* and friends).

Visual Overview: the TextPage Dictionary Structure



6.21.1.1 Page Dictionary

Key	Value
width	width of the clip rectangle (<i>float</i>)
height	height of the clip rectangle (<i>float</i>)
blocks	<i>list</i> of block dictionaries

6.21.1.2 Block Dictionaries

Block dictionaries come in two different formats for **image blocks** and for **text blocks**.

- (Changed in v1.18.0) – new dict key *number*, the block number.
- (Changed in v1.18.11) – new dict key *transform*, the image transformation matrix for image blocks.
- (Changed in v1.18.11) – new dict key *size*, the size of the image in bytes for image blocks.

Image block:

Key	Value
type	1 = image (<i>int</i>)
bbox	image bbox on page (<i>rect_like</i>)
number	block count (<i>int</i>)
ext	image type (<i>str</i>), as file extension, see below
width	original image width (<i>int</i>)
height	original image height (<i>int</i>)
colorspace	colorspace component count (<i>int</i>)
xres	resolution in x-direction (<i>int</i>)
yres	resolution in y-direction (<i>int</i>)
bpc	bits per component (<i>int</i>)
transform	matrix transforming image rect to bbox (<i>matrix_like</i>)
size	size of the image in bytes (<i>int</i>)
image	image content (<i>bytes</i>)

Possible values of the “ext” key are “bmp”, “gif”, “jpeg”, “jpx” (JPEG 2000), “jxr” (JPEG XR), “png”, “pnm”, and “tiff”.

Note:

1. An image block is generated for **all and every image occurrence** on the page. Hence there may be duplicates, if an image is shown at different locations.
2. *TextPage* and corresponding method *Page.get_text()* are **available for all document types**. Only for PDF documents, methods *Document.get_page_images()* / *Page.get_images()* offer some overlapping functionality as far as image lists are concerned. But both lists **may or may not** contain the same items. Any differences are most probably caused by one of the following:
 - “Inline” images (see page 214 of the *Adobe PDF References*) of a PDF page are contained in a textpage, but **do not appear** in *Page.get_images()*.
 - Annotations may also contain images – these will **not appear** in *Page.get_images()*.
 - Image blocks in a textpage are generated for **every** image location – whether or not there are any duplicates. This is in contrast to *Page.get_images()*, which will list each image only once (per reference name).
 - Images mentioned in the page’s *object* definition will **always** appear in *Page.get_images()*¹. But it may happen, that there is no “display” command in the page’s *contents* (erroneously or on purpose). In this case the image will **not appear** in the textpage.
3. The image’s “transformation matrix” is defined as the matrix, for which the expression `bbox / transform == fitz.Rect(0, 0, 1, 1)` is true, lookup details here: *Image Transformation Matrix*.

¹ Image specifications for a PDF page are done in a page’s (sub-) *dictionary*, called “/Resources”. Resource dictionaries can be **inherited** from the page’s parent object (usually the *catalog*). The PDF creator may e.g. define one /Resources on file level, naming all images and all fonts ever used by any page. In these cases, *Page.get_images()* and *Page.get_fonts()* will return the same lists for all pages.

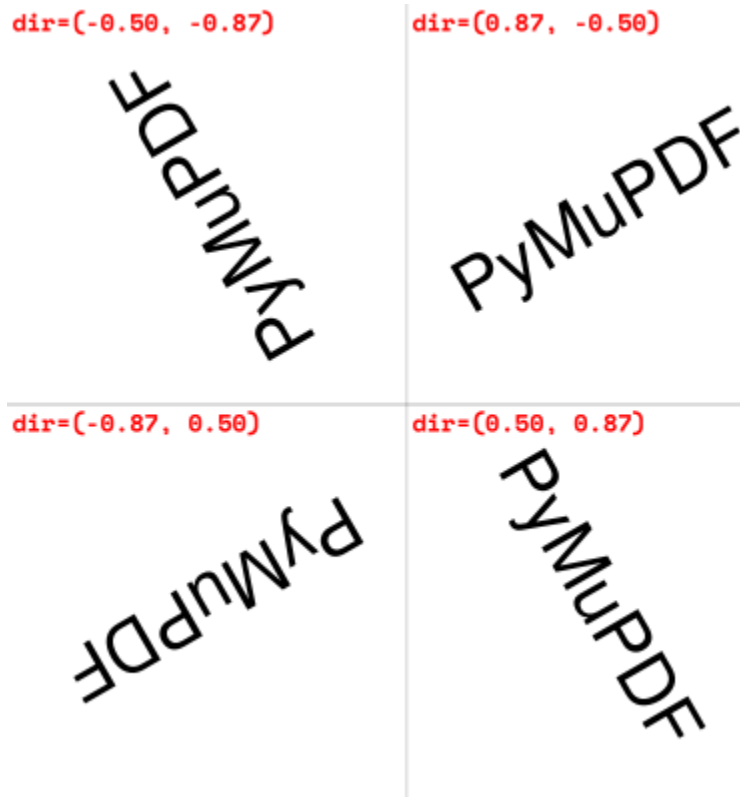
Text block:

Key	Value
type	0 = text (<i>int</i>)
bbox	block rectangle, <i>rect_like</i>
number	block count (<i>int</i>)
lines	<i>list</i> of text line dictionaries

6.21.1.3 Line Dictionary

Key	Value
bbox	line rectangle, <i>rect_like</i>
wmode	writing mode (<i>int</i>): 0 = horizontal, 1 = vertical
dir	writing direction, <i>point_like</i>
spans	<i>list</i> of span dictionaries

The value of key “*dir*” is the **unit vector** $\text{dir} = (\cosine, \sin)$ of the angle, which the text has relative to the x-axis. See the following picture: The word in each quadrant (counter-clockwise from top-right to bottom-right) is rotated by 30, 120, 210 and 300 degrees respectively.



6.21.1.4 Span Dictionary

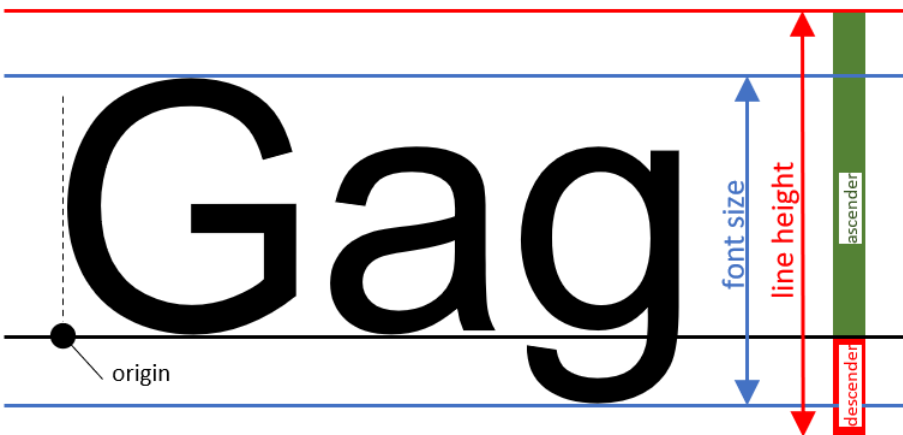
Spans contain the actual text. A line contains **more than one span only**, if it contains text with different font properties.

- Changed in version 1.14.17 Spans now also have a *bbox* key (again).
- Changed in version 1.17.6 Spans now also have an *origin* key.

Key	Value
bbox	span rectangle, <i>rect_like</i>
origin	the first character's origin, <i>point_like</i>
font	font name (<i>str</i>)
ascender	ascender of the font (<i>float</i>)
descender	descender of the font (<i>float</i>)
size	font size (<i>float</i>)
flags	font characteristics (<i>int</i>)
color	text color in sRGB format (<i>int</i>)
text	(only for <code>extractDICT()</code>) text (<i>str</i>)
chars	(only for <code>extractRAWDICT()</code>) list of character dictionaries

(New in version 1.16.0): “color” is the text color encoded in sRGB (int) format, e.g. 0xFF0000 for red. There are functions for converting this integer back to formats (r, g, b) (PDF with float values from 0 to 1) `sRGB_to_pdf()`, or (R, G, B), `sRGB_to_rgb()` (with integer values from 0 to 255).

(New in v1.18.5): “ascender” and “descender” are font properties, provided relative to fontsize 1. Note that descender is a negative value. The following picture shows the relationship to other values and properties.



These numbers may be used to compute the minimum height of a character (or span) – as opposed to the standard height provided in the “bbox” values (which actually represents the **line height**). The following code recalculates the span bbox to have a height of **fontsize** exactly fitting the text inside:

```
>>> a = span["ascender"]
>>> d = span["descender"]
>>> r = fitz.Rect(span["bbox"])
>>> o = fitz.Point(span["origin"]) # its y-value is the baseline
>>> r.y1 = o.y - span["size"] * d / (a - d)
>>> r.y0 = r.y1 - span["size"]
>>> # r now is a rectangle of height 'fontsize'
```

Caution: The above calculation may deliver a **larger** height! This may e.g. happen for OCR'd documents, where the risk of all sorts of text artifacts is high. MuPDF tries to come up with a reasonable bbox height, independently from the fontsize found in the PDF. So please ensure that the height of `span["bbox"]` is **larger** than `span["size"]`.

Note: You may request PyMuPDF to do all of the above automatically by executing `fitz.TOOLS.set_small_glyph_heights(True)`. This sets a global parameter so that all subsequent text searches and text extractions are based on reduced glyph heights, where meaningful.

The following shows the original span rectangle in red and the rectangle with re-computed height in blue.



“*flags*” is an integer, which represents font properties except for the first bit 0. They are to be interpreted like this:

- bit 0: superscripted (2^0) – not a font property, detected by MuPDF code.
- 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**1: print("italic")
>>> # etc.
```

Bits 1 thru 4 are font properties, i.e. encoded in the font program. Please note, that this information is not necessarily correct or complete: fonts quite often contain wrong data here.

6.21.1.5 Character Dictionary for `extractRAW`DICT()

Key	Value
origin	character's left baseline point, <i>point_like</i>
bbox	character rectangle, <i>rect_like</i>
c	the character (unicode)



This image shows the relationship between a character's bbox and its quad:

6.22 TextWriter

- New in v1.16.18

This class represents a MuPDF *text* object. The basic idea is to **decouple (1) text preparation, and (2) text output** to PDF pages.

During **preparation**, a text writer stores any number of text pieces (“spans”) together with their positions and individual font information. The **output** of the writer’s prepared content may happen multiple times to any PDF page with a compatible page size.

A text writer is an elegant alternative to methods `Page.insert_text()` and friends:

- **Improved text positioning:** Choose any point where insertion of text should start. Storing text returns the “cursor position” after the *last character* of the span.
- **Free font choice:** Each text span has its own font and fontsize. This lets you easily switch when composing a larger text.
- **Automatic fallback fonts:** If a character is not supported by the chosen font, alternative fonts are automatically searched. This significantly reduces the risk of seeing unprintable symbols in the output (“TOFUs” – looking like a small rectangle). PyMuPDF now also comes with the **universal font “Droid Sans Fallback Regular”**, which supports **all Latin** characters (including Cyrillic and Greek), and **all CJK** characters (Chinese, Japanese, Korean).
- **Cyrillic and Greek Support:** The *PDF Base 14 Fonts* have integrated support of Cyrillic and Greek characters **without specifying encoding**. Your text may be a mixture of Latin, Greek and Cyrillic.
- **Transparency support:** Parameter *opacity* is supported. This offers a handy way to create watermark-style text.
- **Justified text:** Supported for any font – not just simple fonts as in `Page.insert_textbox()`.
- **Reusability:** A TextWriter object exists independent from PDF pages. It can be written multiple times, either to the same or to other pages, in the same or in different PDFs, choosing different colors or transparency.

Using this object entails three steps:

1. When **created**, a TextWriter requires a fixed **page rectangle** in relation to which it calculates text positions. A text writer can write to pages of this size only.
2. Store text in the TextWriter using methods `TextWriter.append()`, `TextWriter.appendv()` and `TextWriter.fill_textbox()` as often as is desired.
3. Output the TextWriter object on some PDF page(s).

Note:

- Starting with version 1.17.0, TextWriters **do support** text rotation via the *morph* parameter of `TextWriter.write_text()`.
 - There also exists `Page.write_text()` which combines one or more TextWriters and jointly writes them to a given rectangle and with a given rotation angle – much like `Page.show_pdf_page()`.
-

Method / Attribute	Short Description
<code>append()</code>	Add text in horizontal write mode
<code>appendv()</code>	Add text in vertical write mode
<code>fill_textbox()</code>	Fill rectangle (horizontal write mode)
<code>write_text()</code>	Output TextWriter to a PDF page
<code>color</code>	Text color (can be changed)
<code>last_point</code>	Last written character ends here
<code>opacity</code>	Text opacity (can be changed)
<code>rect</code>	Page rectangle used by this TextWriter
<code>text_rect</code>	Area occupied so far

Class API

class TextWriter

`__init__(self, rect, opacity=1, color=None)`

Parameters

- **rect** (*rect-like*) – rectangle internally used for text positioning computations.
- **opacity** (*float*) – sets the transparency for the text to store here. Values outside the interval `[0, 1]` will be ignored. A value of e.g. 0.5 means 50% transparency.
- **color** (*float, sequ*) – the color of the text. All colors are specified as floats `0 <= color <= 1`. A single float represents some gray level, a sequence implies the colorspace via its length.

`append(pos, text, font=None, fontsize=11, language=None, right_to_left=False, small_caps=0)`

- *Changed in v1.18.9*
- *Changed in v1.18.15*

Add some new text in horizontal writing.

Parameters

- **pos** (*point-like*) – start position of the text, the bottom left point of the first character.
- **text** (*str*) – a string of arbitrary length. It will be written starting at position “pos”.
- **font** – a *Font*. If omitted, `fitz.Font("helv")` will be used.
- **fontsize** (*float*) – the fontsize, a positive number, default 11.
- **language** (*str*) – the language to use, e.g. “en” for English. Meaningful values should be compliant with the ISO 639 standards 1, 2, 3 or 5. Reserved for future use: currently has no effect as far as we know.
- **right_to_left** (*bool*) – (*New in v1.18.9*) whether the text should be written from right to left. Applicable for languages like Arabian or Hebrew. Default is *False*. If *True*, any Latin parts within the text will automatically converted. There are no other consequences, i.e. `TextWriter.last_point` will still be the rightmost character, and there neither is any alignment taking place. Hence you may want to use `TextWriter.fill_textbox()` instead.
- **small_caps** (*bool*) – (*New in v1.18.15*) look for the character’s Small Capital version in the font. If present, take that value instead. Otherwise the original character

(this font or the fallback font) will be taken. The fallback font will never return small caps. For example, this snippet:

```
>>> doc = fitz.open()
>>> page = doc.new_page()
>>> text = "PyMuPDF: the Python bindings for MuPDF"
>>> font = fitz.Font("figo") # choose a font with small caps
>>> tw = fitz.TextWriter(page.rect)
>>> tw.append((50,100), text, font=font, small_caps=True)
>>> tw.write_text(page)
>>> doc.ez_save("x.pdf")
```

PYMUPDF: THE PYTHON BINDINGS FOR MUPDF

will produce this PDF text:

Returns

`text_rect` and `last_point`. (Changed in v1.18.0:) Raises an exception for an unsupported font – checked via `Font.is_writable`.

appendv(*pos*, *text*, *font=None*, *fontsize=11*, *language=None*, *small_caps=0*)

Changed in v1.18.15

Add some new text in vertical, top-to-bottom writing.

Parameters

- **pos** (*point_like*) – start position of the text, the bottom left point of the first character.
- **text** (*str*) – a string. It will be written starting at position “pos”.
- **font** – a *Font*. If omitted, `fitz.Font("helv")` will be used.
- **fontsize** (*float*) – the fontsize, a positive float, default 11.
- **language** (*str*) – the language to use, e.g. “en” for English. Meaningful values should be compliant with the ISO 639 standards 1, 2, 3 or 5. Reserved for future use: currently has no effect as far as we know.
- **small_caps** (*bool*) – (New in v1.18.15) see `append()`.

Returns

`text_rect` and `last_point`. (Changed in v1.18.0:) Raises an exception for an unsupported font – checked via `Font.is_writable`.

fill_textbox(*rect*, *text*, ***, *pos=None*, *font=None*, *fontsize=11*, *align=0*, *right_to_left=False*, *warn=None*, *small_caps=0*)

- Changed in 1.17.3: New parameter *pos* to specify where to start writing within rectangle.
- Changed in v1.18.9: Return list of lines which do not fit in rectangle. Support writing right-to-left (e.g. Arabian, Hebrew).
- Changed in v1.18.15: Prefer small caps if supported by the font.

Fill a given rectangle with text in horizontal writing mode. This is a convenience method to use as an alternative for `append()`.

Parameters

- **rect** (*rect_like*) – the area to fill. No part of the text will appear outside of this.

- **text** (*str, sequ*) – the text. Can be specified as a (UTF-8) string or a list / tuple of strings. A string will first be converted to a list using *splitlines()*. Every list item will begin on a new line (forced line breaks).
- **pos** (*point_like*) – (*new in v1.17.3*) start storing at this point. Default is a point near rectangle top-left.
- **font** – the *Font*, default *fitz.Font("helv")*.
- **fontsize** (*float*) – the fontsize.
- **align** (*int*) – text alignment. Use one of `TEXT_ALIGN_LEFT`, `TEXT_ALIGN_CENTER`, `TEXT_ALIGN_RIGHT` or `TEXT_ALIGN_JUSTIFY`.
- **right_to_left** (*bool*) – (*New in v1.18.9*) whether the text should be written from right to left. Applicable for languages like Arabian or Hebrew. Default is *False*. If *True*, any Latin parts are automatically reverted. You must still set the alignment (if you want right alignment), it does not happen automatically – the other alignment options remain available as well.
- **warn** (*bool*) – on text overflow do nothing, warn, or raise an exception. Overflow text will never be written. **Changed in v1.18.9:**
 - Default is *None*.
 - The list of overflow lines will be returned.
- **small_caps** (*bool*) – (*New in v1.18.15*) see *append()*.

Return type

list

Returns

New in v1.18.9 – List of lines that did not fit in the rectangle. Each item is a tuple (*text*, *length*) containing a string and its length (on the page).

Note: Use these methods as often as is required – there is no technical limit (except memory constraints of your system). You can also mix *append()* and text boxes and have multiple of both. Text positioning is exclusively controlled by the insertion point. Therefore there is no need to adhere to any order. (*Changed in v1.18.0:*) Raise an exception for an unsupported font – checked via *Font.is_writable*.

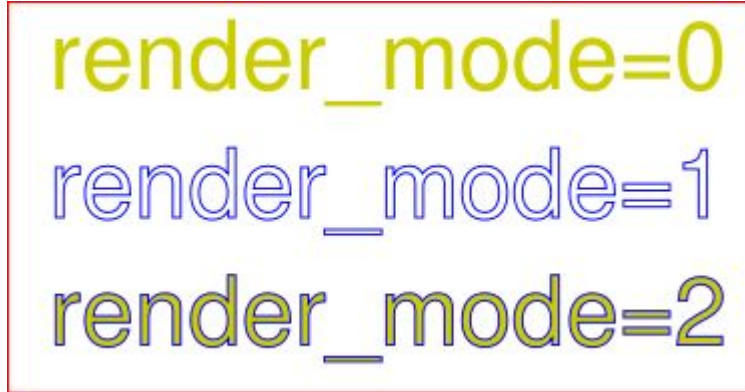
write_text(*page, opacity=None, color=None, morph=None, overlay=True, oc=0, render_mode=0*)

Write the TextWriter text to a page, which is the only mandatory parameter. The other parameters can be used to temporarily override the values used when the TextWriter was created.

Parameters

- **page** – write to this *Page*.
- **opacity** (*float*) – override the value of the TextWriter for this output.
- **color** (*sequ*) – override the value of the TextWriter for this output.
- **morph** (*sequ*) – modify the text appearance by applying a matrix to it. If provided, this must be a sequence (*fixpoint, matrix*) with a point-like *fixpoint* and a matrix-like *matrix*. A typical example is rotating the text around *fixpoint*.
- **overlay** (*bool*) – put in foreground (default) or background.
- **oc** (*int*) – (*new in v1.18.4*) the *xref* of an *OCG* or *OCMD*.

- **render_mode** (*int*) – The PDF Tr operator value. Values: 0 (default), 1, 2, 3



(invisible).

text_rect

The area currently occupied.

Return type

Rect

last_point

The “cursor position” – a *Point* – after the last written character (its bottom-right).

Return type

Point

opacity

The text opacity (modifyable).

Return type

float

color

The text color (modifyable).

Return type

float,tuple

rect

The page rectangle for which this TextWriter was created. Must not be modified.

Return type

Rect

Note: To see some demo scripts dealing with TextWriter, have a look at [this](#) repository.

1. Opacity and color apply to **all the text** in this object.
2. If you need different colors / transparency, you must create a separate TextWriter. Whenever you determine the color should change, simply append the text to the respective TextWriter using the previously returned `last_point` as position for the new text span.
3. Appending items or text boxes can occur in arbitrary order: only the position parameter controls where text appears.
4. Font and fontsize can freely vary within the same TextWriter. This can be used to let text with different properties appear on the same displayed line: just specify *pos* accordingly, and e.g. set it to `last_point` of the previously added item.

5. You can use the *pos* argument of `TextWriter.fill_textbox()` to set the position of the first text character. This allows filling the same textbox with contents from different `TextWriter` objects, thus allowing for multiple colors, opacities, etc.
 6. MuPDF does not support all fonts with this feature, e.g. no Type3 fonts. Starting with v1.18.0 this can be checked via the font attribute `Font.is_writable`. This attribute is also checked when using `TextWriter` methods.
-

6.23 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 ¹
<code>Tools.mupdf_warnings()</code>	return the accumulated MuPDF warnings
<code>Tools.mupdf_display_errors()</code>	return the accumulated MuPDF warnings
<code>Tools.reset_mupdf_warnings()</code>	empty MuPDF messages on STDOUT
<code>Tools.set_aa_level()</code>	set the anti-aliasing values
<code>Tools.set_annot_stem()</code>	set the prefix of new annotation / link ids
<code>Tools.set_small_glyph_heights()</code>	search and extract using small bbox heights
<code>Tools.set_subset_fontnames()</code>	control suppression of subset fontname tags
<code>Tools.show_aa_level()</code>	return the anti-aliasing values
<code>Tools.unset_quad_corrections()</code>	disable PyMuPDF-specific code
<code>Tools.fitz_config</code>	configuration settings of PyMuPDF
<code>Tools.store_maxsize</code>	maximum storables cache size
<code>Tools.store_size</code>	current storables cache size

Class API

`class Tools`

`gen_id()`

A convenience method returning a unique positive integer which will increase by 1 on every invocation. Example usages include creating unique keys in databases - its creation should be faster than using timestamps by an order of magnitude.

Note: MuPDF has dropped support for this in v1.14.0, so we have re-implemented a similar function with the following differences:

- It is not part of MuPDF's global context and not threadsafe (not an issue because we do not support threads in PyMuPDF anyway).
 - It is implemented as *int*. This means that the maximum number is `sys.maxsize`. Should this number ever be exceeded, the counter starts over again at 1.
-

¹ This memory area is internally used by MuPDF, and it serves as a cache for objects that have already been read and interpreted, thus improving performance. The most bulky object types are images and also fonts. When an application starts up the MuPDF library (in our case this happens as part of `import fitz`), it must specify a maximum size for this area. PyMuPDF's uses the default value (256 MB) to limit memory consumption. Use the methods here to control or investigate store usage. For example: even after a document has been closed and all related objects have been deleted, the store usage may still not drop down to zero. So you might want to enforce that before opening another document.

Return type

int

Returns

a unique positive integer.

set_annot_stem(*stem=None*)

- New in v1.18.6

Set or inquire the prefix for the id of new annotations, fields or links.

Parameters

stem (*str*) – if omitted, the current value is returned, default is “fitz”. Annotations, fields / widgets and links technically are subtypes of the same type of object (*/Annot*) in PDF documents. An */Annot* object may be given a unique identifier within a page. For each of the applicable subtypes, PyMuPDF generates identifiers “stem-Annn”, “stem-Wnnn” or “stem-Lnnn” respectively. The number “nnn” is used to enforce the required uniqueness.

Return type

str

Returns

the current value.

set_small_glyph_heights(*on=None*)

- New in v1.18.5

Set or inquire reduced bbox heights in text extract and text search methods.

Parameters

on (*bool*) – if omitted or *None*, the current setting is returned. For other values the *bool()* function is applied to set a global variable. If *True*, [Page.search_for\(\)](#) and [Page.get_text\(\)](#) methods return character, span, line or block bboxes that have a height of *font size*. If *False* (standard setting when PyMuPDF is imported), bbox height will be based on font properties and normally equal *line height*.

Return type

bool

Returns*True* or *False*.

Note: Text extraction options “xml”, “xhtml” and “html”, which directly wrap MuPDF code, are not influenced by this.

set_subset_fontnames(*on=None*)

- New in v1.18.9

Control suppression of subset fontname tags in text extractions.

Parameters

on (*bool*) – if omitted / *None*, the current setting is returned. Arguments evaluating to *True* or *False* set a global variable. If *True*, options “dict”, “json”, “rawdict” and “rawjson” will return e.g. “NOHSJV+Calibri-Light”, otherwise only “Calibri-Light” (the default). The setting remains in effect until changed again.

Return type

bool

Returns*True* or *False*.

Note: Except mentioned above, no other text extraction variants are influenced by this. This is especially true for the options “xml”, “xhtml” and “html”, which are based on MuPDF code. They extract the font name “Calibri-Light”, or even just the **family** name – Calibri in this example.

unset_quad_corrections(*on=None*)

- New in v1.18.10

Enable / disable PyMuPDF-specific code, that tries to rebuild valid character quads when encountering nonsense in `Page.get_text()` text extractions. This code depends on certain font properties (ascender and descender), which do not exist in rare situations and cause segmentation faults when trying to access them. This method sets a global parameter in PyMuPDF, which suppresses execution of this code.

Parameters

on (*bool*) – if omitted or *None*, the current setting is returned. For other values the *bool()* function is applied to set a global variable. If *True*, PyMuPDF will not try to access the resp. font properties and use values `ascender=0.8` and `descender=-0.2` instead.

Return type

bool

Returns*True* or *False*.**store_shrink**(*percent*)

Reduce the storables cache by a percentage of its current size.

Parameters

percent (*int*) – the percentage of current size to free. If 100+ the store will be emptied, if zero, nothing will happen. MuPDF’s caching strategy is “least recently used”, so low-usage elements get deleted first.

Return type

int

Returns

the new current store size. Depending on the situation, the size reduction may be larger than the requested percentage.

show_aa_level()

- New in version 1.16.14

Return the current anti-aliasing values. These values control the rendering quality of graphics and text elements.

Return type

dict

Returns

A dictionary with the following initial content: `{'graphics': 8, 'text': 8, 'graphics_min_line_width': 0.0}`.

set_aa_level(*level*)

- New in version 1.16.14

Set the new number of bits to use for anti-aliasing. The same value is taken currently for graphics and text rendering. This might change in a future MuPDF release.

Parameters

level (*int*) – an integer ranging between 0 and 8. Value outside this range will be silently changed to valid values. The value will remain in effect throughout the current session or until changed again.

reset_mupdf_warnings()

- New in version 1.16.0

Empty MuPDF warnings message buffer.

mupdf_display_errors(*value=None*)

- New in version 1.16.8

Show or set whether MuPDF errors should be displayed.

Parameters

value (*bool*) – if not a bool, the current setting is returned. If true, MuPDF errors will be shown on *sys.stderr*, otherwise suppressed. In any case, messages continue to be stored in the warnings store. Upon import of PyMuPDF this value is *True*.

Returns

True or *False*

mupdf_warnings(*reset=True*)

- New in version 1.16.0

Return all stored MuPDF messages as a string with interspersed line-breaks.

Parameters

reset (*bool*) – (*new in version 1.16.7*) whether to automatically empty the store.

fitz_config

A dictionary containing the actual values used for configuring PyMuPDF and MuPDF. Also refer to the installation chapter. This is an overview of the keys, each of which describes the status of a support aspect.

Key	Support included for ...
plotter-g	Gray colorspace rendering
plotter-rgb	RGB colorspace rendering
plotter-cmyk	CMYK colorspace rendering
plotter-n	overprint rendering
pdf	PDF documents
xps	XPS documents
svg	SVG documents
cbz	CBZ documents
img	IMG documents
html	HTML documents
epub	EPUB documents
jpx	JPEG2000 images
js	JavaScript
tofu	all TOFU fonts
tofu-cjk	CJK font subset (China, Japan, Korea)
tofu-cjk-ext	CJK font extensions
tofu-cjk-lang	CJK font language extensions
tofu-emoji	TOFU emoji fonts
tofu-historic	TOFU historic fonts
tofu-symbol	TOFU symbol fonts
tofu-sil	TOFU SIL fonts
icc	ICC profiles
py-memory	using Python memory management ²
base14	Base-14 fonts (should always be true)

For an explanation of the term “TOFU” see [this Wikipedia article](#)..

```
In [1]: import fitz
In [2]: TOOLS.fitz_config
Out[2]:
{'plotter-g': True,
 'plotter-rgb': True,
 'plotter-cmyk': True,
 'plotter-n': True,
 'pdf': True,
 'xps': True,
 'svg': True,
 'cbz': True,
 'img': True,
 'html': True,
 'epub': True,
 'jpx': True,
 'js': True,
 'tofu': False,
 'tofu-cjk': True,
 'tofu-cjk-ext': False,
 'tofu-cjk-lang': False,
 'tofu-emoji': False,
```

(continues on next page)

² By default PyMuPDF and MuPDF use `malloc()/free()` for dynamic memory management. One can instead force them to use the Python allocation functions `PyMem_New()/PyMem_Del()`, by modifying `fitz/fitz.i` to do `#define JM_MEMORY 1` and rebuilding PyMuPDF.

(continued from previous page)

```
'tofu-historic': False,
'tofu-symbol': False,
'tofu-sil': False,
'icc': True,
'py-memory': False,
'base14': True}
```

Return type

dict

store_maxsize

Maximum storables cache size in bytes. PyMuPDF is generated with a value of 268'435'456 (256 MB, the default value), which you should therefore always see here. If this value is zero, then an “unlimited” growth is permitted.

Return type

int

store_size

Current storables cache size in bytes. This value may change (and will usually increase) with every use of a PyMuPDF function. It will (automatically) decrease only when `Tools.store_maxsize` is going to be exceeded: in this case, MuPDF will evict low-usage objects until the value is again in range.

Return type

int

6.23.1 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].get_pixmap(alpha=False)
>>> fitz.TOOLS.store_size
454519
# release (at least) 50% of the storage
>>> fitz.TOOLS.store_shrink(50)
13471
>>> fitz.TOOLS.store_size
13471
# get a few unique numbers
>>> fitz.TOOLS.gen_id()
1
>>> fitz.TOOLS.gen_id()
2
```

(continues on next page)

(continued from previous page)

```
>>> fitz.TOOLS.gen_id()
3
# close document and see how much cache is still in use
>>> doc.close()
>>> fitz.TOOLS.store_size
0
>>>
```

6.24 Widget

This class represents a PDF Form field, also called a “widget”. Throughout this documentation, we are using these terms synonymously. Fields technically are a special case of PDF annotations, which allow users with limited permissions to enter information in a PDF. This is primarily used for filling out forms.

Like annotations, widgets live on PDF pages. Similar to annotations, the first widget on a page is accessible via [Page.first_widget](#) and subsequent widgets can be accessed via the [Widget.next](#) property.

(*Changed in version 1.16.0*) MuPDF no longer treats widgets as a subset of general annotations. Consequently, [Page.first_annot](#) and [Annot.next\(\)](#) will deliver **non-widget annotations exclusively**, and be *None* if only form fields exist on a page. Vice versa, [Page.first_widget](#) and [Widget.next\(\)](#) will only show widgets. This design decision is purely internal to MuPDF; technically, links, annotations and fields have a lot in common and also continue to share the better part of their code within (Py-) MuPDF.

Class API

class Widget

button_states()

New in version 1.18.15

Return the names of On / Off (i.e. selected / clicked or not) states a button field may have. While the ‘Off’ state usually is also named like so, the ‘On’ state is often given a name relating to the functional context, for example ‘Yes’, ‘Female’, etc.

This method helps finding out the possible values of [field_value](#) in these cases.

returns

a dictionary with the names of ‘On’ and ‘Off’ for the *normal* and the *pressed-down* appearance of button widgets. Example:

```
>>> print(field.field_name, field.button_states())
Gender Second person {'down': ['Male', 'Off'], 'normal': [
↪ 'Male', 'Off']}
```

update()

After any changes to a widget, this method **must be used** to store them in the PDF¹.

reset()

Reset the field’s value to its default – if defined – or remove it. Do not forget to issue [update\(\)](#) afterwards.

¹ If you intend to re-access a new or updated field (e.g. for making a pixmap), make sure to reload the page first. Either close and re-open the document, or load another page first, or simply do `page = doc.reload_page(page)`.

next

Point to the next form field on the page. The last widget returns *None*.

border_color

A list of up to 4 floats defining the field's border color. Default value is *None* which causes border style and border width to be ignored.

border_style

A string defining the line style of the field's border. See [Annot.border](#). Default is "s" ("Solid") – a continuous line. Only the first character (upper or lower case) will be regarded when creating a widget.

border_width

A float defining the width of the border line. Default is 1.

border_dashes

A list/tuple of integers defining the dash properties of the border line. This is only meaningful if *border_style* == "D" and *border_color* is provided.

choice_values

Python sequence of strings defining the valid choices of list boxes and combo boxes. For these widget types, this property is mandatory and must contain at least two items. Ignored for other types.

field_name

A mandatory string defining the field's name. No checking for duplicates takes place.

field_label

An optional string containing an "alternate" field name. Typically used for any notes, help on field usage, etc. Default is the field name.

field_value

The value of the field.

field_flags

An integer defining a large amount of properties of a field. Be careful when changing this attribute as this may change the field type.

field_type

A mandatory integer defining the field type. This is a value in the range of 0 to 6. It cannot be changed when updating the widget.

field_type_string

A string describing (and derived from) the field type.

fill_color

A list of up to 4 floats defining the field's background color.

button_caption

The caption string of a button-type field.

is_signed

A bool indicating the signing status of a signature field, else *None*.

rect

The rectangle containing the field.

text_color

A list of **1, 3 or 4 floats** defining the text color. Default value is black (*[0, 0, 0]*).

text_font

A string defining the font to be used. Default and replacement for invalid values is “*Helv*”. For valid font reference names see the table below.

text_fontsize

A float defining the text fontsize. Default value is zero, which causes PDF viewer software to dynamically choose a size suitable for the annotation’s rectangle and text amount.

text_maxlen

An integer defining the maximum number of text characters. PDF viewers will (should) not accept a longer text.

text_type

An integer defining acceptable text types (e.g. numeric, date, time, etc.). For reference only for the time being – will be ignored when creating or updating widgets.

xref

The PDF [xref](#) of the widget.

script

- New in version 1.16.12

JavaScript text (unicode) for an action associated with the widget, or *None*. This is the only script action supported for **button type** widgets.

script_stroke

- New in version 1.16.12

JavaScript text (unicode) to be performed when the user types a key-stroke into a text field or combo box or modifies the selection in a scrollable list box. This action can check the keystroke for validity and reject or modify it. *None* if not present.

script_format

- New in version 1.16.12

JavaScript text (unicode) to be performed before the field is formatted to display its current value. This action can modify the field’s value before formatting. *None* if not present.

script_change

- New in version 1.16.12

JavaScript text (unicode) to be performed when the field’s value is changed. This action can check the new value for validity. *None* if not present.

script_calc

- New in version 1.16.12

JavaScript text (unicode) to be performed to recalculate the value of this field when that of another field changes. *None* if not present.

Note:

1. For **adding** or **changing** one of the above scripts, just put the appropriate JavaScript source code in the widget attribute. To **remove** a script, set the respective attribute to *None*.
 2. Button fields only support [script](#). Other script entries will automatically be set to *None*.
-

6.24.1 Standard Fonts for Widgets

Widgets use their own resources object */DR*. A widget resources object must at least contain a */Font* object. Widget fonts are independent from page fonts. We currently support the 14 PDF base fonts using the following fixed reference names, or any name of an already existing field font. When specifying a text font for new or changed widgets, **either** choose one in the first table column (upper and lower case supported), **or** one of the already existing form fonts. In the latter case, spelling must exactly match.

To find out already existing field fonts, inspect the list `Document.FormFonts`.

Reference	Base14 Fontname
CoBI	Courier-BoldOblique
CoBo	Courier-Bold
CoIt	Courier-Oblique
Cour	Courier
HeBI	Helvetica-BoldOblique
HeBo	Helvetica-Bold
HeIt	Helvetica-Oblique
Helv	Helvetica (default)
Symb	Symbol
TiBI	Times-BoldItalic
TiBo	Times-Bold
TiIt	Times-Italic
TiRo	Times-Roman
ZaDb	ZapfDingbats

You are generally free to use any font for every widget. However, we recommend using *ZaDb* (“ZapfDingbats”) and fontsize 0 for check boxes: typical viewers will put a correctly sized tickmark in the field’s rectangle, when it is clicked.

6.24.2 Supported Widget Types

PyMuPDF supports the creation and update of many, but not all widget types.

- text (PDF_WIDGET_TYPE_TEXT)
- push button (PDF_WIDGET_TYPE_BUTTON)
- check box (PDF_WIDGET_TYPE_CHECKBOX)
- combo box (PDF_WIDGET_TYPE_COMBOBOX)
- list box (PDF_WIDGET_TYPE_LISTBOX)
- radio button (PDF_WIDGET_TYPE_RADIOBUTTON): PyMuPDF does not currently support groups of (interconnected) buttons, where setting one automatically unsets the other buttons in the group. The widget object also does not reflect the presence of a button group. Setting or unsetting happens via values `True` and `False` and will always work without affecting other radio buttons.
- signature (PDF_WIDGET_TYPE_SIGNATURE) **read only**.

6.25 Xml

- New in v1.21.0

This represents an HTML or an XML node. It is a helper class intended to access the DOM (Document Object Model) content of a *Story* object.

There is no need to ever directly construct an *Xml* object: after creating a *Story*, simply take *Story.body* – which is an *Xml* node – and use it to navigate your way through the story’s DOM.

Method / Attribute	Description
<code>add_bullet_list()</code>	add a <i>ul</i> tag - bulleted list, context manager.
<code>add_codeblock()</code>	add a <i>pre</i> tag, context manager.
<code>add_description_list()</code>	add a <i>dl</i> tag, context manager.
<code>add_division()</code>	add a <i>div</i> tag (renamed from “section”), context manager.
<code>add_header()</code>	add a header tag (one of <i>h1</i> to <i>h6</i>), context manager.
<code>add_horizontal_line()</code>	add a <i>hr</i> tag.
<code>add_image()</code>	add a <i>img</i> tag.
<code>add_link()</code>	add a <i>a</i> tag.
<code>add_number_list()</code>	add a <i>ol</i> tag, context manager.
<code>add_paragraph()</code>	add a <i>p</i> tag.
<code>add_span()</code>	add a <i>span</i> tag, context manager.
<code>add_subscript()</code>	add subscript text (<i>sub</i> tag) - inline element, treated like text.
<code>add_superscript()</code>	add subscript text (<i>sup</i> tag) - inline element, treated like text.
<code>add_code()</code>	add code text (<i>code</i> tag) - inline element, treated like text.
<code>add_var()</code>	add code text (<i>code</i> tag) - inline element, treated like text.
<code>add_samp()</code>	add code text (<i>code</i> tag) - inline element, treated like text.
<code>add_kbd()</code>	add code text (<i>code</i> tag) - inline element, treated like text.
<code>add_text()</code>	add a text string. Line breaks <code>\n</code> are honored as <i>br</i> tags.
<code>set_align()</code>	sets the alignment using a CSS style spec. Only works for block-level tags.
<code>set_attribute()</code>	sets an arbitrary key to some value (which may be empty).
<code>set_bgcolor()</code>	sets the background color. Only works for block-level tags.
<code>set_bold()</code>	sets bold on or off or to some string value.
<code>set_color()</code>	sets text color.
<code>set_columns()</code>	sets the number of columns. Argument may be any valid number or string.
<code>set_font()</code>	sets the font-family, e.g. “sans-serif”.
<code>set_fontsize()</code>	sets the font size. Either a float or a valid HTML/CSS string.
<code>set_id()</code>	sets a <i>id</i> . A check for uniqueness is performed.
<code>set_italic()</code>	sets italic on or off or to some string value.
<code>set_leading()</code>	set inter-block text distance (<code>-mupdf-leading</code>), only works on block-level nodes.
<code>set_lineheight()</code>	set height of a line. Float like 1.5, which sets to $1.5 * \text{fontsize}$.
<code>set_margins()</code>	sets the margin(s), float or string with up to 4 values.
<code>set_pagebreak_after()</code>	insert a page break after this node.
<code>set_pagebreak_before()</code>	insert a page break before this node.
<code>set_properties()</code>	set any or all desired properties in one call.
<code>add_style()</code>	set (add) some “style” attribute not supported by its own <code>set_</code> method.
<code>add_class()</code>	set (add) some “class” attribute.
<code>set_text_indent()</code>	set indentation for first textblock line. Only works for block-level nodes.
<code>tagname</code>	either the HTML tag name like <i>p</i> or <code>None</code> if a text node.
<code>text</code>	either the node’s text or <code>None</code> if a tag node.
<code>is_text</code>	check if the node is a text.
<code>first_child</code>	contains the first node one level below this one (or <code>None</code>).

continues on next page

Table 5 – continued from previous page

Method / Attribute	Description
<code>last_child</code>	contains the last node one level below this one (or None).
<code>next</code>	the next node at the same level (or None).
<code>previous</code>	the previous node at the same level.
<code>root</code>	the top node of the DOM, which hence has the tagname <i>html</i> .

Class API**class Xml****add_bullet_list()**

Add an *ul* tag - bulleted list, context manager. See [ul](#).

add_codeblock()

Add a *pre* tag, context manager. See [pre](#).

add_description_list()

Add a *dl* tag, context manager. See [dl](#).

add_division()

Add a *div* tag, context manager. See [div](#).

add_header(value)

Add a header tag (one of *h1* to *h6*), context manager. See [headings](#).

Parameters

value (*int*) – a value 1 - 6.

add_horizontal_line()

Add a *hr* tag. See [hr](#).

add_image(name, width=None, height=None)

Add an *img* tag. This causes the inclusion of the named image in the DOM.

Parameters

- **name** (*str*) – the filename of the image. This **must be the member name** of some entry of the [Archive](#) parameter of the [Story](#) constructor.
- **width** – if provided, either an absolute (int) value, or a percentage string like “30%”. A percentage value refers to the width of the specified **where** rectangle in [Story.place\(\)](#). If this value is provided and **height** is omitted, the image will be included keeping its aspect ratio.
- **height** – if provided, either an absolute (int) value, or a percentage string like “30%”. A percentage value refers to the height of the specified **where** rectangle in [Story.place\(\)](#). If this value is provided and **width** is omitted, the image’s aspect ratio will be honored.

add_link(href, text=None)

Add an *a* tag - inline element, treated like text.

Parameters

- **href** (*str*) – the URL target.
- **text** (*str*) – the text to display. If omitted, the **href** text is shown instead.

add_number_list()

Add an *ol* tag, context manager.

add_paragraph()

Add a *p* tag, context manager.

add_span()

Add a *span* tag, context manager. See [span](#)

add_subscript(*text*)

Add “subscript” text (*sub* tag) - inline element, treated like text.

add_superscript(*text*)

Add “superscript” text (*sup* tag) - inline element, treated like text.

add_code(*text*)

Add “code” text (*code* tag) - inline element, treated like text.

add_var(*text*)

Add “variable” text (*var* tag) - inline element, treated like text.

add_samp(*text*)

Add “sample output” text (*samp* tag) - inline element, treated like text.

add_kbd(*text*)

Add “keyboard input” text (*kbd* tag) - inline element, treated like text.

add_text(*text*)

Add a text string. Line breaks `\n` are honored as *br* tags.

set_align(*value*)

Set the text alignment. Only works for block-level tags.

Parameters

value – either one of the [Text Alignment](#) or the `text-align` values.

set_attribute(*key*, *value=None*)

Set an arbitrary key to some value (which may be empty).

Parameters

- **key** (*str*) – the name of the attribute.
- **value** (*str*) – the (optional) value of the attribute.

get_attributes()

Retrieve all attributes of the current nodes as a dictionary.

Returns

a dictionary with the attributes and their values of the node.

get_attribute_value(*key*)

Get the attribute value of *key*.

Parameters

key (*str*) – the name of the attribute.

Returns

a string with the value of *key*.

remove_attribute(*key*)

Remove the attribute **key** from the node.

Parameters

key (*str*) – the name of the attribute.

set_bgcolor(*value*)

Sets the background color. Only works for block-level tags.

Parameters

value – either an RGB value like (255, 0, 0) (for “red”) or a valid [background-color](#) value.

set_bold(*value*)

Sets bold on or off or to some string value.

Parameters

value – True, False or a valid [font-weight](#) value.

set_color(*value*)

Set the color of the text following.

Parameters

value – either an RGB value like (255, 0, 0) (for “red”) or a valid [color](#) value.

set_columns(*value*)

Sets the number of columns.

Parameters

value – a valid [columns](#) value.

Note: Currently ignored - supported in a future MuPDF version.

set_font(*value*)

Set the font-family.

Parameters

value (*str*) – e.g. “sans-serif”.

set_fontsize(*value*)

Set the font size for text following.

Parameters

value – a float or a valid [font-size](#) value.

set_id(*unqid*)

Sets a *id*. This serves as a unique identification of the node within the DOM. Use it to easily locate the node to inspect or modify it. A check for uniqueness is performed.

Parameters

unqid (*str*) – id string of the node.

set_italic(*value*)

Sets italic on or off or to some string value for the text following it.

Parameters

value – True, False or some valid [font-style](#) value.

set_leading(*value*)

Set inter-block text distance (`-mupdf-leading`), only works on block-level nodes.

Parameters

value (*float*) – the distance in points to the previous block.

set_lineheight(*value*)

Set height of a line.

Parameters

value – a float like 1.5 (which sets to $1.5 * \text{fontsize}$), or some valid [line-height](#) value.

set_margins(*value*)

Sets the margin(s).

Parameters

value – float or string with up to 4 values. See [CSS documentation](#).

set_pagebreak_after()

Insert a page break after this node.

set_pagebreak_before()

Insert a page break before this node.

set_properties(*align=None, bgcolor=None, bold=None, color=None, columns=None, font=None, fontsize=None, indent=None, italic=None, leading=None, lineheight=None, margins=None, pagebreak_after=False, pagebreak_before=False, unqid=None, cls=None*)

Set any or all desired properties in one call. The meaning of argument values equal the values of the corresponding `set_` methods.

Note: The properties set by this method are directly attached to the node, whereas every `set_` method generates a new *span* below the current node that has the respective property. So to e.g. “globally” set some property for the *body*, this method must be used.

add_style(*value*)

Set (add) some style attribute not supported by its own `set_` method.

Parameters

value (*str*) – any valid CSS style value.

add_class(*value*)

Set (add) some “class” attribute.

Parameters

value (*str*) – the name of the class. Must have been defined in either the HTML or the CSS source of the DOM.

set_text_indent(*value*)

Set indentation for the first textblock line. Only works for block-level nodes.

Parameters

value – a valid [text-indent](#) value. Please note that negative values do not work.

append_child(*node*)

Append a child node. This is a low-level method used by other methods like [Xml.add_paragraph\(\)](#).

Parameters

node – the *Xml* node to append.

create_text_node(text)

Create direct text for the current node

Parameters

text (*str*) – the text to append.

Return type

Xml

Returns

the created element.

create_element(tag)

Create a new node with a given tag. This is a low-level method used by other methods like *Xml.add_paragraph()*.

Parameters

tag (*str*) – the element tag.

Return type

Xml

Returns

the created element. To actually bind it to the DOM, use *Xml.append_child()*.

insert_before(elem)

Insert the given element *elem* before this node.

Parameters

elem – some *Xml* element.

insert_after(elem)

Insert the given element *elem* after this node.

Parameters

elem – some *Xml* element.

clone()

Make a copy of this node, which then may be appended (using *Xml.append_child()*) or inserted (using one of *Xml.insert_before()*, *Xml.insert_after()*) in this DOM.

Returns

the clone (*Xml*) of the current node.

remove()

Remove this node from the DOM.

debug()

For debugging purposes, print this node's structure in a simplified form.

find(tag, att, match)

Under the current node, find a node with the given *tag*, attribute *att* and value *match*.

Parameters

- **tag** (*str*) – restrict search to this tag. May be *None* for unrestricted search.
- **att** (*str*) – check this attribute.
- **match** (*str*) – the desired attribute value to match.

Return type*Xml*.**Returns**

None if nothing found, otherwise the first matching node.

find_next(*tag, att, match*)Continue a previous *Xml.find()* with the same values.**Return type***Xml*.**Returns**

None if none more found, otherwise the next matching node.

tagnameEither the HTML tag name like *p* or None if a text node.**text**

Either the node's text or None if a tag node.

is_text

Check if a text node.

first_child

Contains the first node one level below this one (or None).

last_child

Contains the last node one level below this one (or None).

next

The next node at the same level (or None).

previous

The previous node at the same level.

rootThe top node of the DOM, which hence has the tagname *html*.

6.25.1 Setting Text properties

In HTML tags can be nested such that innermost text **inherits properties** from the tag enveloping its parent tag. For example `<p>some bold text<i>this is bold and italic</i>regular text</p>`.

To achieve the same effect, methods like *Xml.set_bold()* and *Xml.set_italic()* each open a temporary *span* with the desired property underneath the current node.

In addition, these methods return there parent node, so they can be concatenated with each other.

6.25.2 Context Manager support

The standard way to add nodes to a DOM is this:

```
body = story.body
para = body.add_paragraph() # add a paragraph
para.set_bold() # text that follows will be bold
para.add_text("some bold text")
para.set_italic() # text that follows will additionally be italic
para.add_text("this is bold and italic")
para.set_italic(False).set_bold(False) # all following text will be regular
para.add_text("regular text")
```

Methods that are flagged as “context managers” can conveniently be used in this way:

```
body = story.body
with body.add_paragraph() as para:
    para.set_bold().add_text("some bold text")
    para.set_italic().add_text("this is bold and italic")
    para.set_italic(False).set_bold(False).add_text("regular text")
para.add_text("more regular text")
```


OPERATOR ALGEBRA FOR GEOMETRY OBJECTS

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

They all are special cases of Python sequences, see *Using Python Sequences as Arguments in PyMuPDF* for more background.

We have defined operators for these classes that allow dealing with them (almost) like ordinary numbers in terms of addition, subtraction, multiplication, division, and some others.

This chapter is a synopsis of what is possible.

7.1 General Remarks

1. Operators can be either **binary** (i.e. involving two objects) or **unary**.
2. The resulting type of **binary** operations is either a **new object of the left operand’s class** or a bool.
3. The result of **unary** operations is either a **new object** of the same class, a bool or a float.
4. The binary operators `+`, `-`, `*`, `/` are defined for all classes. They *roughly* do what you would expect – **except, that the second operand ...**
 - may always be a number which then performs the operation on every component of the first one,
 - may always be a numeric sequence of the same length (2, 4 or 6) – we call such sequences *point_like*, *rect_like*, *quad_like* or *matrix_like*, respectively.
5. Rectangles support additional binary operations: **intersection** (operator `&`), **union** (operator `|`) and **containment** checking.
6. Binary operators fully support in-place operations, so expressions like `a /= b` are valid if `b` is numeric or “a_like”.

7.2 Unary Operations

Oper.	Result
<code>bool(OBJ)</code>	is false exactly if all components of OBJ are zero
<code>abs(OBJ)</code>	the rectangle area – equal to <code>norm(OBJ)</code> for the other tyes
<code>norm(OBJ)</code>	square root of the component squares (Euclidean norm)
<code>+OBJ</code>	new copy of OBJ
<code>-OBJ</code>	new copy of OBJ with negated components
<code>~m</code>	inverse of matrix “m”, or the null matrix if not invertible

7.3 Binary Operations

For every geometry object “a” and every number “b”, the operations “a ° b” and “a ° = b” are always defined for the operators +, -, *, /. The respective operation is simply executed for each component of “a”. If the **second operand is not a number**, then the following is defined:

Oper.	Result
a+b, a-b	component-wise execution, “b” must be “a-like”.
a*m, a/m	“a” can be a point, rectangle or matrix, but “m” must be <i>matrix_like</i> . “a/m” is treated as “a*~m” (see note below for non-invertible matrices). If “a” is a point or a rectangle , then “a.transform(m)” is executed. If “a” is a matrix, then matrix concatenation takes place.
a&b	intersection rectangle : “a” must be a rectangle and “b” <i>rect_like</i> . Delivers the largest rectangle contained in both operands.
a b	union rectangle : “a” must be a rectangle, and “b” may be <i>point_like</i> or <i>rect_like</i> . Delivers the smallest rectangle containing both operands.
b in a	if “b” is a number, then <code>b in tuple(a)</code> is returned. If “b” is <i>point_like</i> , <i>rect_like</i> or <i>quad_like</i> , then “a” must be a rectangle, and <code>a.contains(b)</code> is returned.
a == b	<i>True</i> if <code>bool(a-b)</code> is <i>False</i> (“b” may be “a-like”).

Note: Please note an important difference to usual arithmetics:

Matrix multiplication is **not commutative**, i.e. in general we have $m*n \neq n*m$ for two matrices. Also, there are non-zero matrices which have no inverse, for example `m = Matrix(1, 0, 1, 0, 1, 0)`. If you try to divide by any of these, you will receive a `ZeroDivisionError` exception using operator “/”, e.g. for the expression `fitz.Identity / m`. But if you formulate `fitz.Identity * ~m`, the result will be `fitz.Matrix()` (the null matrix).

Admittedly, this represents an inconsistency, and we are considering to remove it. For the time being, you can choose to avoid an exception and check whether `~m` is the null matrix, or accept a potential `ZeroDivisionError` by using `fitz.Identity / m`.

Note:

- With these conventions, all the usual algebra rules apply. For example, arbitrarily using brackets (**among objects of the same class!**) is possible: if `r1`, `r2` are rectangles and `m1`, `m2` are matrices, you can do this `(r1 + r2) * m1 * m2`.
- For all objects of the same class, `a + b + c == (a + b) + c == a + (b + c)` is true.
- For matrices in addition the following is true: `(m1 + m2) * m3 == m1 * m3 + m2 * m3` (distributivity property).
- **But the sequence of applying matrices is important: If `r` is a rectangle and `m1`, `m2` are matrices, then – caution!:**

$$- r * m1 * m2 == (r * m1) * m2 \neq r * (m1 * m2)$$

7.4 Some Examples

7.4.1 Manipulation with numbers

For the usual arithmetic operations, numbers are always allowed as second operand. In addition, you can formulate "`x in OBJ`", where `x` is a number. It is implemented as "`x in tuple(OBJ)`":

```
>>> fitz.Rect(1, 2, 3, 4) + 5
fitz.Rect(6.0, 7.0, 8.0, 9.0)
>>> 3 in fitz.Rect(1, 2, 3, 4)
True
>>>
```

The following will create the upper left quarter of a document page rectangle:

```
>>> page.rect
Rect(0.0, 0.0, 595.0, 842.0)
>>> page.rect / 2
Rect(0.0, 0.0, 297.5, 421.0)
>>>
```

The following will deliver the **middle point of a line** that connects two points **p1** and **p2**:

```
>>> p1 = fitz.Point(1, 2)
>>> p2 = fitz.Point(4711, 3141)
>>> mp = (p1 + p2) / 2
>>> mp
Point(2356.0, 1571.5)
>>>
```

7.4.2 Manipulation with “like” Objects

The second operand of a binary operation can always be “like” the left operand. “Like” in this context means “a sequence of numbers of the same length”. With the above examples:

```
>>> p1 + p2
Point(4712.0, 3143.0)
>>> p1 + (4711, 3141)
Point(4712.0, 3143.0)
>>> p1 += (4711, 3141)
>>> p1
Point(4712.0, 3143.0)
>>>
```

To shift a rectangle for 5 pixels to the right, do this:

```
>>> fitz.Rect(100, 100, 200, 200) + (5, 0, 5, 0) # add 5 to the x coordinates
Rect(105.0, 100.0, 205.0, 200.0)
>>>
```

Points, rectangles and matrices can be *transformed* with matrices. In PyMuPDF, we treat this like a “**multiplication**” (or resp. “**division**”), where the second operand may be “like” a matrix. Division in this context means “multiplication with the inverted matrix”:

```

>>> m = fitz.Matrix(1, 2, 3, 4, 5, 6)
>>> n = fitz.Matrix(6, 5, 4, 3, 2, 1)
>>> p = fitz.Point(1, 2)
>>> p * m
Point(12.0, 16.0)
>>> p * (1, 2, 3, 4, 5, 6)
Point(12.0, 16.0)
>>> p / m
Point(2.0, -2.0)
>>> p / (1, 2, 3, 4, 5, 6)
Point(2.0, -2.0)
>>>
>>> m * n # matrix multiplication
Matrix(14.0, 11.0, 34.0, 27.0, 56.0, 44.0)
>>> m / n # matrix division
Matrix(2.5, -3.5, 3.5, -4.5, 5.5, -7.5)
>>>
>>> m / m # result is equal to the Identity matrix
Matrix(1.0, 0.0, 0.0, 1.0, 0.0, 0.0)
>>>
>>> # look at this non-invertible matrix:
>>> m = fitz.Matrix(1, 0, 1, 0, 1, 0)
>>> ~m
Matrix(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
>>> # we try dividing by it in two ways:
>>> p = fitz.Point(1, 2)
>>> p * ~m # this delivers point (0, 0):
Point(0.0, 0.0)
>>> p / m # but this is an exception:
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    p / m
  File "... /site-packages/fitz/fitz.py", line 869, in __truediv__
    raise ZeroDivisionError("matrix not invertible")
ZeroDivisionError: matrix not invertible
>>>

```

As a specialty, rectangles support additional binary operations:

- **intersection** – the common area of rectangle-likes, operator “&”
- **inclusion** – enlarge to include a point-like or rect-like, operator “|”
- **containment** check – whether a point-like or rect-like is inside

Here is an example for creating the smallest rectangle enclosing given points:

```

>>> # first define some point-likes
>>> points = []
>>> for i in range(10):
    for j in range(10):
        points.append((i, j))
>>>
>>> # now create a rectangle containing all these 100 points
>>> # start with an empty rectangle

```

(continues on next page)

(continued from previous page)

```
>>> r = fitz.Rect(points[0], points[0])
>>> for p in points[1:]: # and include remaining points one by one
    r |= p
>>> r # here is the to be expected result:
Rect(0.0, 0.0, 9.0, 9.0)
>>> (4, 5) in r # this point-like lies inside the rectangle
True
>>> # and this rect-like is also inside
>>> (4, 4, 5, 5) in r
True
>>>
```


LOW LEVEL FUNCTIONS AND CLASSES

Contains a number of functions and classes for the experienced user. To be used for special needs or performance requirements.

8.1 Functions

The following are miscellaneous functions and attributes on a fairly low-level technical detail.

Some functions provide detail access to PDF structures. Others are stripped-down, high performance versions of other functions which provide more information.

Yet others are handy, general-purpose utilities.

Function	Short Description
<code>Annot.apn_bbox</code>	PDF only: bbox of the appearance object
<code>Annot.apn_matrix</code>	PDF only: the matrix of the appearance object
<code>Page.is_wrapped</code>	check whether contents wrapping is present
<code>adobe_glyph_names()</code>	list of glyph names defined in Adobe Glyph List
<code>adobe_glyph_unicodes()</code>	list of unicodes defined in Adobe Glyph List
<code>Annot.clean_contents()</code>	PDF only: clean the annot's <i>contents</i> object
<code>Annot.set_apn_bbox()</code>	PDF only: set the bbox of the appearance object
<code>Annot.set_apn_matrix()</code>	PDF only: set the matrix of the appearance object
<code>ConversionHeader()</code>	return header string for <i>get_text</i> methods
<code>ConversionTrailer()</code>	return trailer string for <i>get_text</i> methods
<code>Document.del_xml_metadata()</code>	PDF only: remove XML metadata
<code>Document.get_char_widths()</code>	PDF only: return a list of glyph widths of a font
<code>Document.get_new_xref()</code>	PDF only: create and return a new <i>xref</i> entry
<code>Document.is_stream()</code>	PDF only: check whether an <i>xref</i> is a stream object
<code>Document.xml_metadata_xref()</code>	PDF only: return XML metadata <i>xref</i> number
<code>Document.xref_length()</code>	PDF only: return length of <i>xref</i> table
<code>EMPTY_IRECT()</code>	return the (standard) empty / invalid rectangle
<code>EMPTY_QUAD()</code>	return the (standard) empty / invalid quad
<code>EMPTY_RECT()</code>	return the (standard) empty / invalid rectangle
<code>get_pdf_now()</code>	return the current timestamp in PDF format
<code>get_pdf_str()</code>	return PDF-compatible string
<code>get_text_length()</code>	return string length for a given font & fontsize
<code>glyph_name_to_unicode()</code>	return unicode from a glyph name
<code>image_profile()</code>	return a dictionary of basic image properties
<code>INFINITE_IRECT()</code>	return the (only existing) infinite rectangle

continues on next page

Table 1 – continued from previous page

Function	Short Description
<code>INFINITE_QUAD()</code>	return the (only existing) infinite quad
<code>INFINITE_RECT()</code>	return the (only existing) infinite rectangle
<code>make_table()</code>	split rectangle in sub-rectangles
<code>Page.clean_contents()</code>	PDF only: clean the page's <i>contents</i> objects
<code>Page.get_bboxlog()</code>	list of rectangles that envelop text, drawing or image objects
<code>Page.get_contents()</code>	PDF only: return a list of content <i>xref</i> numbers
<code>Page.get_displaylist()</code>	create the page's display list
<code>Page.get_text_blocks()</code>	extract text blocks as a Python list
<code>Page.get_text_words()</code>	extract text words as a Python list
<code>Page.get_texttrace()</code>	low-level text information
<code>Page.read_contents()</code>	PDF only: get complete, concatenated /Contents source
<code>Page.run()</code>	run a page through a device
<code>Page.set_contents()</code>	PDF only: set page's <i>contents</i> to some <i>xref</i>
<code>Page.wrap_contents()</code>	wrap contents with stacking commands
<code>css_for_pymupdf_font()</code>	create CSS source for a font in package <code>pymupdf_fonts</code>
<code>paper_rect()</code>	return rectangle for a known paper format
<code>paper_size()</code>	return width, height for a known paper format
<code>paper_sizes()</code>	dictionary of pre-defined paper formats
<code>planish_line()</code>	matrix to map a line to the x-axis
<code>recover_char_quad()</code>	compute the quad of a char ("rawdict")
<code>recover_line_quad()</code>	compute the quad of a subset of line spans
<code>recover_quad()</code>	compute the quad of a span ("dict", "rawdict")
<code>recover_quad()</code>	return the quad for a text span ("dict" / "rawdict")
<code>recover_span_quad()</code>	compute the quad of a subset of span characters
<code>sRGB_to_pdf()</code>	return PDF RGB color tuple from an sRGB integer
<code>sRGB_to_rgb()</code>	return (R, G, B) color tuple from an sRGB integer
<code>unicode_to_glyph_name()</code>	return glyph name from a unicode
<code>fitz_fontdescriptors</code>	dictionary of available supplement fonts
<code>TESSDATA_PREFIX</code>	a copy of <code>os.environ["TESSDATA_PREFIX"]</code>
<code>pdfcolor</code>	dictionary of almost 500 RGB colors in PDF format.

paper_size(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 in 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.paper_size("A4")` returns (595, 842) and `fitz.paper_size("letter-l")` delivers (792, 612).

paper_rect(*s*)

Convenience function to return a [Rect](#) for a known paper format.

Parameters

s (*str*) – any format name supported by [paper_size\(\)](#).

Return type

[Rect](#)

Returns

fitz.Rect(0, 0, width, height) with *width, height=fitz.paper_size(s)*.

```
>>> import fitz
>>> fitz.paper_rect("letter-l")
fitz.Rect(0.0, 0.0, 792.0, 612.0)
>>>
```

sRGB_to_pdf(*srgb*)

New in v1.17.4

Convenience function returning a PDF color triple (red, green, blue) for a given sRGB color integer as it occurs in [Page.get_text\(\)](#) dictionaries “dict” and “rawdict”.

Parameters

srgb (*int*) – an integer of format RRGGBB, where each color component is an integer in range(255).

Returns

a tuple (red, green, blue) with float items in interval $0 \leq \text{item} \leq 1$ representing the same color. Example `sRGB_to_pdf(0xff0000) = (1, 0, 0)` (red).

sRGB_to_rgb(*srgb*)

New in v1.17.4

Convenience function returning a color (red, green, blue) for a given *sRGB* color integer.

Parameters

srgb (*int*) – an integer of format RRGGBB, where each color component is an integer in range(255).

Returns

a tuple (red, green, blue) with integer items in range(256) representing the same color. Example `sRGB_to_pdf(0xff0000) = (255, 0, 0)` (red).

glyph_name_to_unicode(*name*)

New in v1.18.0

Return the unicode number of a glyph name based on the **Adobe Glyph List**.

Parameters

name (*str*) – the name of some glyph. The function is based on the [Adobe Glyph List](#).

Return type

int

Returnsthe unicode. Invalid *name* entries return `0xfffd` (65533).

Note: A similar functionality is provided by package `fontTools` in its *agl* sub-package.

unicode_to_glyph_name(*ch*)*New in v1.18.0*Return the glyph name of a unicode number, based on the **Adobe Glyph List**.**Parameters****ch** (*int*) – the unicode given by e.g. `ord("ß")`. The function is based on the [Adobe Glyph List](#).**Return type**

str

Returnsthe glyph name. E.g. `fitz.unicode_to_glyph_name(ord("Ä"))` returns `'Adieresis'`.

Note: A similar functionality is provided by package `fontTools` in its *agl* sub-package.

adobe_glyph_names()*New in v1.18.0*Return a list of glyph names defined in the **Adobe Glyph List**.**Return type**

list

Returns

list of strings.

Note: A similar functionality is provided by package `fontTools` in its *agl* sub-package.

adobe_glyph_unicodes()*New in v1.18.0*Return a list of unicodes for there exists a glyph name in the **Adobe Glyph List**.**Return type**

list

Returns

list of integers.

Note: A similar functionality is provided by package `fontTools` in its *agl* sub-package.

`css_for_pymupdf_font(fontcode, *, CSS=None, archive=None, name=None)`

New in v1.21.0

Utility function for use with “Story” applications.

Create CSS @font-face items for the given fontcode in pymupdf-fonts. Creates a CSS font-family for all fonts starting with string “fontcode”.

The font naming convention in package pymupdf-fonts is “fontcode<sf>”, where the suffix “sf” is one of “” (empty), “it”/”i”, “bo”/”b” or “bi”. These suffixes thus represent the regular, italic, bold or bold-italic variants of that font.

For example, font code “notos” refers to fonts

- “notos” - “Noto Sans Regular”
- “notosit” - “Noto Sans Italic”
- “notosbo” - “Noto Sans Bold”
- “notosbi” - “Noto Sans Bold Italic”

The function creates (up to) four CSS @font-face definitions and collectively assigns the font-family name “notos” to them (or the “name” value if provided). Associated font buffers are placed / added to the provided archive.

To use the font in the Python API for *Story*, execute `.set_font(fontcode)` (or “name” if given). The correct font weight or style will automatically be selected as required.

For example to replace the “sans-serif” HTML standard (i.e. Helvetica) with the above “notos”, execute the following. Whenever “sans-serif” is used (whether explicitly or implicitly), the Noto Sans fonts will be selected.

```
CSS = fitz.css_for_pymupdf_font("notos", name="sans-serif", archive=...)
```

Expects and returns the CSS source, with the new CSS definitions appended.

Parameters

- **fontcode** (*str*) – one of the font codes present in package `pymupdf-fonts` (usually) representing the regular version of the font family.
- **CSS** (*str*) – any already existing CSS source, or `None`. The function will append its new definitions to this. This is the string that **must be used** as `user_css` when creating the *Story*.
- **archive** – *Archive*, **mandatory**. All font binaries (i.e. up to four) found for “fontcode” will be added to the archive. This is the archive that **must be used** as `archive` when creating the *Story*.
- **name** (*str*) – the name under which the “fontcode” fonts should be found. If omitted, “fontcode” will be used.

Return type

`str`

Returns

Modified CSS, with appended @font-face statements for each font variant of fontcode. Fontbuffers associated with “fontcode” will have been added

to ‘archive’. The function will automatically find up to 4 font variants. All pymupdf-fonts (that are no special purpose like math or music, etc.) have regular, bold, italic and bold-italic variants. To see currently available font codes check `fitz.fitz_fontdescriptors.keys()`. This will show something like `dict_keys(['cascadia', 'cascadiai', 'cascadiab', 'cascadiabi', 'figbo', 'figo', 'figbi', 'figit', 'fimbo', 'fimo', 'spacembo', 'spacembi', 'spacemit', 'spacemo', 'math', 'music', 'symbol1', 'symbol2', 'notosbo', 'notosbi', 'notosit', 'notos', 'ubuntu', 'ubuntubo', 'ubuntubi', 'ubuntuit', 'ubuntm', 'ubuntmbo', 'ubuntmbi', 'ubuntmit'])`.

Here is a complete snippet for using the “Noto Sans” font instead of “Helvetica”:

```
arch = fitz.Archive()
CSS = fitz.css_for_pymupdf_font("notos", name="sans-serif", archive=arch)
story = fitz.Story(user_css=CSS, archive=arch)
```

recover_quad(*line_dir*, *span*)

New in v1.18.9

Convenience function returning the quadrilateral envelopping the text of a text span, as returned by [Page.get_text\(\)](#) using the “dict” or “rawdict” options.

Parameters

- **line_dict** (*tuple*) – the value `line["dir"]` of the span’s line.
- **span** (*dict*) – the span sub-dictionary.

Returns

the quadrilateral of the span’s text.

make_table(*rect*, *cols=1*, *rows=1*)

New in v1.17.4

Convenience function to split a rectangle into sub-rectangles. Returns a list of *rows* lists, each containing *cols* [Rect](#) items. Each sub-rectangle can then be addressed by its row and column index.

Parameters

- **rect** (*rect_like*) – the rectangle to split.
- **cols** (*int*) – the desired number of columns.
- **rows** (*int*) – the desired number of rows.

Returns

a list of [Rect](#) objects of equal size, whose union equals *rect*. Here is the layout of a 3x4 table created by `cell = fitz.make_table(rect, cols=4, rows=3)`:

<code>cell[0][0]</code>	<code>cell[0][1]</code>	<code>cell[0][2]</code>	<code>cell[0][3]</code>
<code>cell[1][0]</code>	<code>cell[1][1]</code>	<code>cell[1][2]</code>	<code>cell[1][3]</code>
<code>cell[2][0]</code>	<code>cell[2][1]</code>	<code>cell[2][2]</code>	<code>cell[2][3]</code>

planish_line(p1, p2)

- New in version 1.16.2)*

Return a matrix which maps the line from p1 to p2 to the x-axis such that p1 will become (0,0) and p2 a point with the same distance to (0,0).

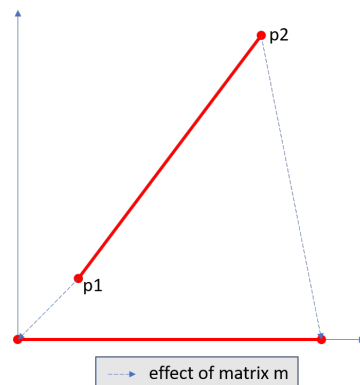
Parameters

- **p1** (*point_like*) – starting point of the line.
- **p2** (*point_like*) – end point of the line.

Return type*Matrix***Returns**

a matrix which combines a rotation and a translation:

```
>>> p1 = fitz.Point(1, 1)
>>> p2 = fitz.Point(4, 5)
>>> abs(p2 - p1) # distance of points
5.0
>>> m = fitz.planish_line(p1, p2)
>>> p1 * m
Point(0.0, 0.0)
>>> p2 * m
Point(5.0, -5.960464477539063e-08)
>>> # distance of the resulting points
>>> abs(p2 * m - p1 * m)
5.0
```

**paper_sizes()**

A dictionary of pre-defines paper formats. Used as basis for *paper_size()*.

fitz_fontdescriptors

- New in v1.17.5

A dictionary of usable fonts from repository *pymupdf-fonts*. Items are keyed by their reserved font-name and provide information like this:

```
In [2]: fitz.fitz_fontdescriptors.keys()
Out[2]: dict_keys(['figbo', 'figo', 'figbi', 'figit', 'fimbo', 'fimo',
'spacembo', 'spacembi', 'spacemit', 'spacemo', 'math', 'music', 'symbol1',
'symbol2'])
In [3]: fitz.fitz_fontdescriptors["fimo"]
Out[3]:
{'name': 'Fira Mono Regular',
'size': 125712,
'mono': True,
'bold': False,
'italic': False,
'serif': True,
'glyphs': 1485}
```

If `pymupdf-fonts` is not installed, the dictionary is empty.

The dictionary keys can be used to define a *Font* via e.g. `font = fitz.Font("fimo")` – just like you can do it with the builtin fonts “Helvetica” and friends.

TESSDATA_PREFIX

- New in v1.19.4

Copy of `os.environ["TESSDATA_PREFIX"]` for convenient checking whether there is integrated Tesseract OCR support.

If this attribute is `None`, Tesseract-OCR is either not installed, or the environment variable is not set to point to Tesseract’s language support folder.

Note: This variable is now checked before OCR functions are tried. This prevents verbose messages from MuPDF.

pdfcolor

- New in v1.19.6

Contains about 500 RGB colors in PDF format with the color name as key. To see what is there, you can obviously look at `fitz.pdfcolor.keys()`.

Examples:

- `fitz.pdfcolor["red"] = (1.0, 0.0, 0.0)`
 - `fitz.pdfcolor["skyblue"] = (0.5294117647058824, 0.807843137254902, 0.9215686274509803)`
 - `fitz.pdfcolor["wheat"] = (0.9607843137254902, 0.8705882352941177, 0.7019607843137254)`
-

get_pdf_now()

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.

get_text_length(text, fontname='helv', fontsize=11, encoding=TEXT_ENCODING_LATIN)

- New in version 1.14.7

Calculate the length of text on output with a given **builtin** font, fontsize and encoding.

Parameters

- **text** (*str*) – the text string.
- **fontname** (*str*) – the fontname. Must be one of either the *PDF Base 14 Fonts* or the CJK fonts, identified by their “reserved” fontnames (see table in :meth:`Page.insert_font`).
- **fontsize** (*float*) – the fontsize.
- **encoding** (*int*) – the encoding to use. Besides 0 = Latin, 1 = Greek and 2 = Cyrillic (Russian) are available. Relevant for Base-14 fonts “Helvetica”, “Courier” and “Times” and their variants only. Make sure to use the same value as in the corresponding text insertion.

Return type

float

Returns

the length in points the string will have (e.g. when used in *Page.insert_text()*).

Note: This function will only do the calculation – it won’t insert font nor text.

Note: The *Font* class offers a similar method, *Font.text_length()*, which supports Base-14 fonts and any font with a character map (CMap, Type 0 fonts).

Warning: If you use this function to determine the required rectangle width for the (*Page* or *Shape*) *insert_textbox* methods, be aware that they calculate on a **by-character level**. Because of rounding effects, this will mostly lead to a slightly larger number: *sum([fitz.get_text_length(c) for c in text]) > fitz.get_text_length(text)*. So either (1) do the same, or (2) use something like *fitz.get_text_length(text + “”)* for your calculation.

get_pdf_str(text)

Make a PDF-compatible string: if the text contains code points $\text{ord}(c) > 255$, then it will be converted to UTF-16BE with BOM as a hexadecimal character string enclosed in "<>" brackets like <feff...>. Otherwise, it will return the string enclosed in (round) brackets, replacing any characters outside the ASCII range with some special code. Also, every "(", ")", or backslash is escaped with a backslash.

Parameters

text (*str*) – the object to convert

Return type

str

Returns

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

image_profile(stream)

- New in v1.16.7
- Changed in v1.19.5: also return natural image orientation extracted from EXIF data if present.

Show important properties of an image provided as a memory area. Its main purpose is to avoid using other Python packages just to determine them.

Parameters

stream (*bytes* / *bytearray* / *BytesIO* / *file*) – an image either in memory or an **opened** file. A memory resident image maybe any of the formats *bytes*, *bytearray* or *io.BytesIO*.

Return type

dict

Returns

No exception is ever raised: in case of error, the empty dictionary {} is returned. Otherwise, there are the following items:

```
In [2]: fitz.image_profile(open("nur-ruhig.jpg", "rb").
↳ read())
Out[2]:
{'width': 439,
'height': 501,
'orientation': 0, # natural orientation (from EXIF)
'transform': (1.0, 0.0, 0.0, 1.0, 0.0, 0.0), # orientation_
↳ matrix
'xres': 96,
'yres': 96,
'colorspace': 3,
'bpc': 8,
'ext': 'jpeg',
'cs-name': 'DeviceRGB'}
```

There is the following relation to *Exif* information encoded in *orientation*, and correspondingly in the *transform* matrix-like (quoted from MuPDF documentation, *ccw* = counter-clockwise):

0. Undefined

1. 0 degree ccw rotation. (Exif = 1)
2. 90 degree ccw rotation. (Exif = 8)
3. 180 degree ccw rotation. (Exif = 3)
4. 270 degree ccw rotation. (Exif = 6)
5. flip on X. (Exif = 2)
6. flip on X, then rotate ccw by 90 degrees. (Exif = 5)
7. flip on X, then rotate ccw by 180 degrees. (Exif = 4)
8. flip on X, then rotate ccw by 270 degrees. (Exif = 7)

Note:

- For some “exotic” images (FAX encodings, RAW formats and the like), this method will not work and return *None*. You can however still work with such images in PyMuPDF, e.g. by using [Document.extract_image\(\)](#) or create pixmaps via `Pixmap(doc, xref)`. These methods will automatically convert exotic images to the PNG format before returning results.
 - You can also get the properties of images embedded in a PDF, via their [xref](#). In this case make sure to extract the raw stream: `fitz.image_profile(doc.xref_stream_raw(xref))`.
 - Images as returned by the image blocks of [Page.get_text\(\)](#) using “dict” or “rawdict” options are also supported.
-

ConversionHeader(*text*, filename=*UNKNOWN*)

Return the header string required to make a valid document out of page text outputs.

Parameters

- **output** (*str*) – type of document. Use the same as the output parameter of [get_text\(\)](#).
- **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.get_text\(\)](#) for an example.

Parameters

- **output** (*str*) – type of document. Use the same as the output parameter of [get_text\(\)](#).

Return type

str

Document.del_xml_metadata()

Delete an object containing XML-based metadata from the PDF. (Py-) MuPDF does not support XML-based metadata. Use this if you want to make sure that the conventional metadata dictionary will be used exclusively. Many thirdparty PDF programs insert their own metadata in XML format and thus may override what you store in the conventional dictionary. This method deletes any such reference, and the corresponding PDF object will be deleted during next garbage collection of the file.

Document.xml_metadata_xref()

Return the XML-based metadata *xref* of the PDF if present – also refer to *Document.del_xml_metadata()*. You can use it to retrieve the content via *Document.xref_stream()* and then work with it using some XML software.

Return type

int

Returns

xref of PDF file level XML metadata – or 0 if none exists.

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.get_bboxlog()

- New in v1.19.0

Returns

a list of rectangles that envelop text, image or drawing objects. Each item is a tuple (*type*, (*x0*, *y0*, *x1*, *y1*)) where the second tuple consists of rectangle coordinates, and *type* is one of the following values:

- "fill-text" – normal text (painted without character borders)
- "stroke-text" – text showing character borders only
- "ignore-text" – text that should not be displayed (e.g. as used by OCR text layers)
- "fill-path" – drawing with fill color (and no border)
- "stroke-path" – drawing with border (and no fill color)
- "fill-image" – displays an image
- "fill-shade" – display a shading

The item sequence represents the **sequence in which these commands are executed** to build the page's appearance. Therefore, if an item's bbox intersects or contains that of a previous item, then the previous item may be (partially) covered / hidden.

So this list is useful to detect such situations. An item's index in this list equals the value of "seqno" keys you will find in the dictionaries returned by `Page.get_drawings()` and `Page.get_texttrace()`.

`Page.get_texttrace()`

- New in v1.18.16
- Changed in v1.19.0: added key "seqno".
- Changed in v1.19.1: stroke and fill colors now always are either RGB or GRAY
- Changed in v1.19.3: span and character bboxes are now also correct if `dir != (1, 0)`.

Return low-level text information of the page. The method is available for **all** document types. The result is a list of Python dictionaries with the following content:

```
{
  'ascender': 0.83251953125,          # font ascender (1)
  'bbox': (458.14019775390625,        # span bbox x0 (7)
          749.4671630859375,        # span bbox y0
          467.76458740234375,        # span bbox x1
          757.5071411132812),        # span bbox y1
  'bidi': 0,                          # bidirectional level (1)
  'chars': (                          # char information, tuple[tuple]
            (45,                      # unicode (4)
             16,                      # glyph id (font dependent)
             (458.14019775390625,      # origin.x (1)
              755.3758544921875),      # origin.y (1)
             (458.14019775390625,      # char bbox x0 (6)
              749.4671630859375,      # char bbox y0
              462.9649963378906,      # char bbox x1
              757.5071411132812)),    # char bbox y1
            ( ... ),                  # more characters
          ),
  'color': (0.0,),                    # text color, tuple[float] (1)
  'colorspace': 1,                    # number of colorspace components
  → (1)
  'descender': -0.30029296875,        # font descender (1)
  'dir': (1.0, 0.0),                  # writing direction (1)
  'flags': 12,                        # font flags (1)
  'font': 'CourierNewPSMT',           # font name (1)
  'linewidth': 0.4019999980926514,    # current line width value (3)
  'opacity': 1.0,                     # alpha value of the text (5)
  'seqno': 246,                       # sequence number (8)
  'size': 8.039999961853027,          # font size (1)
  'spacewidth': 4.824785133358091,    # width of space char
  'type': 0,                          # span type (2)
  'wmode': 0                          # writing mode (1)
}
```

Details:

1. Information above tagged with “(1)” has the same meaning and value as explained in [TextPage](#).
 - Please note that the font `flags` value will never contain a *superscript* flag bit: the detection of superscripts is done within MuPDF [TextPage](#) code – it is not a property of any font.
 - Also note, that the text `color` is encoded as the usual tuple of floats $0 \leq f \leq 1$ – not in sRGB format. Depending on `span["type"]`, interpret this as fill color or stroke color.
2. There are 3 text span types:
 - 0: Filled text – equivalent to PDF text rendering mode 0 (`0 Tr`, the default in PDF), only each character’s “inside” is shown.
 - 1: Stroked text – equivalent to `1 Tr`, only the character borders are shown.
 - 3: Ignored text – equivalent to `3 Tr` (hidden text).
3. Line width in this context is important only for processing `span["type"] != 0`: it determines the thickness of the character’s border line. This value may not be provided at all with the text data. In this case, a value of 5% of the fontsize (`span["size"] * 0,05`) is generated. Often, an “artificial” bold text in PDF is created by `2 Tr`. There is no equivalent span type for this case. Instead, respective text is represented by two consecutive spans – which are identical in every aspect, except for their types, which are 0, resp 1. It is your responsibility to handle this type of situation - in [Page.get_text\(\)](#), MuPDF is doing this for you.
4. For data compactness, the character’s unicode is provided here. Use built-in function `chr()` for the character itself.
5. The alpha / opacity value of the span’s text, $0 \leq \text{opacity} \leq 1$, 0 is invisible text, 1 (100%) is intransparent. Depending in `span["type"]`, interpret this value as *fill* opacity or, resp. *stroke* opacity.
6. (*Changed in v1.19.0*) This value is equal or close to `char["bbox"]` of “rawdict”. In particular, the `bbox` **height** value is always computed as if “**small glyph heights**” had been requested.
7. (*New in v1.19.0*) This is the union of all character bboxes.
8. (*New in v1.19.0*) Enumerates the commands that build up the page’s appearance. Can be used to find out whether text is effectively hidden by objects, which are painted “later”, or *over* some object. So if there is a drawing or image with a higher sequence number, whose `bbox` overlaps (parts of) this text span, one may assume that such an object hides the resp. text. Different text spans have identical sequence numbers if they were created in one go.

Here is a list of similarities and differences of `page.get_texttrace()` compared to `page.get_text("rawdict")`:

- The method is up to **twice as fast**, compared to “rawdict” extraction. Depends on the amount of text.
- The returned data is very **much smaller in size** – although it provides more information.
- Additional types of text **invisibility can be detected**: opacity = 0 or type > 1 or overlapping `bbox` of an object with a higher sequence number.
- If MuPDF returns unicode 0xFFFFD (65533) for unrecognized characters, you may still be able to deduct desired information from the glyph id.
- The `span["chars"]` **contains no spaces**, except the document creator has explicitly coded them. They **will never be generated** like it happens in [Page.get_text\(\)](#) methods. To provide some help for doing your own computations here, the width of a space character is

given. This value is derived from the font where possible. Otherwise the value of a fallback font is taken.

- There is no effort to organize text like it happens for a *TextPage* (the hierarchy of blocks, lines, spans, and characters). Characters are simply extracted in sequence, one by one, and put in a span. Whenever any of the span's characteristics changes, a new span is started. So you may find characters with different `origin.y` values in the same span (which means they would appear in different lines). You cannot assume, that span characters are sorted in any particular order – you must make sense of the info yourself, taking `span["dir"]`, `span["wmode"]`, etc. into account.

- **Ligatures are represented like this:**

- MuPDF handles the following ligatures: “fi”, “ff”, “fl”, “ft”, “st”, “ffi”, and “ffl” (only the first 3 are mostly ever used). If the page contains e.g. ligature “fi”, you will find the following two character items subsequent to each other:

```
(102, glyph, (x, y), (x0, y0, x1, y1)) # 102 = ord("f")
(105, -1, (x, y), (x0, y0, x0, y1))    # 105 = ord("i"),  
→ empty bbox!
```

- This means that the bbox of the first ligature character is the area containing the complete, compound glyph. Subsequent ligature components are recognizable by their glyph value -1 and a bbox of width zero.
- You may want to replace those 2 or 3 char tuples by one, that represents the ligature itself. Use the following mapping of ligatures to unicodes:

```
* "ff" -> 0xFB00
* "fi" -> 0xFB01
* "fl" -> 0xFB02
* "ffi" -> 0xFB03
* "ffl" -> 0xFB04
* "ft" -> 0xFB05
* "st" -> 0xFB06
```

So you may want to replace the two example tuples above by the following single one: `(0xFB01, glyph, (x, y), (x0, y0, x1, y1))` (there is usually no need to lookup the correct glyph id for 0xFB01 in the resp. font, but you may execute `font.has_glyph(0xFB01)` and use its return value).

- **Changed in v1.19.3:** Similar to other text extraction methods, the character and span bboxes envelop the character quads. To recover the quads, follow the same methods *recover_quad()*, *recover_char_quad()* or `:meth:recover_span_quad` as explained in *Structure of Dictionary Outputs*. Use either `None` or `span["dir"]` for the writing direction.

Page.wrap_contents()

Put string pair “q”/“Q” before, resp. after a page’s *Contents* object(s) to ensure that any “geometry” changes are **local** only.

Use this method as an alternative, minimalistic version of *Page.clean_contents()*. Its advantage is a small footprint in terms of processing time and impact on the data size of incremental saves. Multiple executions of this method are no problem and have no functional impact: `b"q q contents Q Q"` is treated like `b"q contents Q"`.

Page.is_wrapped

Indicate whether [Page.wrap_contents\(\)](#) may be required for object insertions in standard PDF geometry. Note that this is a quick, basic check only: a value of *False* may still be a false alarm. But nevertheless executing [Page.wrap_contents\(\)](#) will have no negative side effects.

Return type
bool

Page.get_text_blocks(flags=None)

Deprecated wrapper for [TextPage.extractBLOCKS\(\)](#). Use [Page.get_text\(\)](#) with the “blocks” option instead.

Return type
list[tuple]

Page.get_text_words(flags=None)

Deprecated wrapper for [TextPage.extractWORDS\(\)](#). Use [Page.get_text\(\)](#) with the “words” option instead.

Return type
list[tuple]

Page.get_displaylist()

Run a page through a list device and return its display list.

Return type
[DisplayList](#)

Returns
the display list of the page.

Page.get_contents()

PDF only: Retrieve a list of [xref](#) of [contents](#) objects of a page. May be empty or contain multiple integers. If the page is cleaned ([Page.clean_contents\(\)](#)), it will be one entry at most. The “source” of each */Contents* object can be individually read by [Document.xref_stream\(\)](#) using an item of this list. Method [Page.read_contents\(\)](#) in contrast walks through this list and concatenates the corresponding sources into one bytes object.

Return type
list[int]

Page.set_contents(xref)

PDF only: Let the page’s */Contents* key point to this xref. Any previously used contents objects will be ignored and can be removed via garbage collection.

`Page.clean_contents(sanitize=True)`

- Changed in v1.17.6

PDF only: Clean and concatenate all [contents](#) objects associated with this page. “Cleaning” includes syntactical corrections, standardizations and “pretty printing” of the contents stream. Discrepancies between [contents](#) and [resources](#) objects will also be corrected if `sanitize` is true. See [Page.get_contents\(\)](#) for more details.

Changed in version 1.16.0 Annotations are no longer implicitly cleaned by this method. Use [Annot.clean_contents\(\)](#) separately.

Parameters

sanitize (*bool*) – (new in v1.17.6) if true, synchronization between resources and their actual use in the contents object is synchronized. For example, if a font is not actually used for any text of the page, then it will be deleted from the /Resources/Font object.

Warning: This is a complex function which may generate large amounts of new data and render old data unused. It is **not recommended** using it together with the **incremental save** option. Also note that the resulting singleton new /Contents object is **uncompressed**. So you should save to a **new file** using options “`deflate=True, garbage=3`”.

`Page.read_contents()`

New in version 1.17.0. Return the concatenation of all [contents](#) objects associated with the page – without cleaning or otherwise modifying them. Use this method whenever you need to parse this source in its entirety without having to bother how many separate contents objects exist.

Return type

bytes

`Annot.clean_contents(sanitize=True)`

Clean the [contents](#) streams associated with the annotation. This is the same type of action which [Page.clean_contents\(\)](#) performs – just restricted to this annotation.

`Document.get_char_widths(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.insert_text\(\)](#) and [Page.insert_textbox\(\)](#). 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.get_page_fonts(pno)` of page number *pno* and take the first entry of one of the returned list entries.
- **limit** (*int*) – limits the number of returned entries. The default of 256 is enforced for all fonts that only support 1-byte characters, so-called “simple fonts” (checked by this method). All [PDF Base 14 Fonts](#) are simple fonts.

Return type

list

Returns

a list of *limit* tuples. Each character *c* has an entry (*g*, *w*) in this list with an index of *ord(c)*. Entry *g* (integer) of the tuple is the glyph id of the character, and float *w* is its normalized width. The actual width for some fontsize can be calculated as *w * fontsize*. For simple fonts, the *g* entry can always be safely ignored. In all other cases *g* is the basis for graphically representing *c*.

This function calculates the pixel width of a string called *text*:

```
def pixlen(text, widthlist, fontsize):
    try:
        return sum([widthlist[ord(c)] for c in text]) * fontsize
    except IndexError:
        raise ValueError("max. code point found: %i, increase limit" %
                           ord(max(text)))
```

Document.is_stream(*xref*)

- New in version 1.14.14

PDF only: Check whether the object represented by *xref* is a *stream* type. Return is *False* if not a PDF or if the number is outside the valid xref range.

Parameters

xref (*int*) – *xref* number.

Returns

True if the object definition is followed by data wrapped in keyword pair *stream*, *endstream*.

Document.get_new_xref()

Increase the *xref* by one entry and return that number. This can then be used to insert a new object.

Return type

int :returns: the number of the new *xref* entry. Please note, that only a new entry in the PDF's cross reference table is created. At this point, there will not yet exist a PDF object associated with it. To create an (empty) object with this number use `doc.update_xref(xref, "<<>>")`.

Document.xref_length()

Return length of *xref* table.

Return type*int***Returns**

the number of entries in the *xref* table.

recover_quad(*line_dir*, *span*)

Compute the quadrilateral of a text span extracted via options “dict” or “rawdict” of [Page.get_text\(\)](#).

Parameters

- **line_dir** (*tuple*) – line["dir"] of the owning line. Use None for a span from [Page.get_texttrace\(\)](#).
- **span** (*dict*) – the span.

Returns

the [Quad](#) of the span, usable for text marker annotations (‘Highlight’, etc.).

recover_char_quad(*line_dir*, *span*, *char*)

Compute the quadrilateral of a text character extracted via option “rawdict” of [Page.get_text\(\)](#).

Parameters

- **line_dir** (*tuple*) – line["dir"] of the owning line. Use None for a span from [Page.get_texttrace\(\)](#).
- **span** (*dict*) – the span.
- **char** (*dict*) – the character.

Returns

the [Quad](#) of the character, usable for text marker annotations (‘Highlight’, etc.).

recover_span_quad(*line_dir*, *span*, *chars=None*)

Compute the quadrilateral of a subset of characters of a span extracted via option “rawdict” of [Page.get_text\(\)](#).

Parameters

- **line_dir** (*tuple*) – line["dir"] of the owning line. Use None for a span from [Page.get_texttrace\(\)](#).
- **span** (*dict*) – the span.
- **chars** (*list*) – the characters to consider. If omitted, identical to [recoer_span\(\)](#). If given, the selected extraction option must be “rawdict”.

Returns

the [Quad](#) of the selected characters, usable for text marker annotations (‘Highlight’, etc.).

recover_line_quad(*line*, *spans=None*)

Compute the quadrilateral of a subset of spans of a text line extracted via options “dict” or “rawdict” of [Page.get_text\(\)](#).

Parameters

- **line** (*dict*) – the line.
- **spans** (*list*) – a sub-list of line["spans"]. If omitted, the full line quad will be returned.

Returns

the *Quad* of the selected line spans, usable for text marker annotations ('Highlight', etc.).

INFINITE_QUAD()**INFINITE_RECT()****INFINITE_IRECT()**

Return the (unique) infinite rectangle `Rect(-2147483648.0, -2147483648.0, 2147483520.0, 2147483520.0)`, resp. the *IRect* and *Quad* counterparts. It is the largest possible rectangle: all valid rectangles are contained in it.

EMPTY_QUAD()**EMPTY_RECT()****EMPTY_IRECT()**

Return the “standard” empty and invalid rectangle `Rect(2147483520.0, 2147483520.0, -2147483648.0, -2147483648.0)` resp. *quad*. Its top-left and bottom-right point values are reversed compared to the infinite rectangle. It will e.g. be used to indicate empty bboxes in `page.get_text("dict")` dictionaries. There are however infinitely many empty or invalid rectangles.

8.2 Device

The different format handlers (pdf, xps, etc.) interpret pages to a “device”. Devices are the basis for everything that can be done with a page: rendering, text extraction and searching. The device type is determined by the selected construction method.

Class API**class Device****__init__(self, object, clip)**

Constructor for either a pixel map or a display list device.

Parameters

- **object** (*Pixmap* or *DisplayList*) – either a *Pixmap* or a *DisplayList*.
- **clip** (*IRect*) – An optional *IRect* for *Pixmap* devices to restrict rendering to a certain area of the page. If the complete page is required, specify *None*. For display list devices, this parameter must be omitted.

__init__(self, textpage, flags=0)

Constructor for a text page device.

Parameters

- **textpage** (*TextPage*) – *TextPage* object
- **flags** (*int*) – control the way how text is parsed into the text page. Currently 3 options can be coded into this parameter, see *Text Extraction Flags*. To set these options use something like `flags=0 | TEXT_PRESERVE_LIGATURES | ...`

8.3 Working together: DisplayList and TextPage

Here are some instructions on how to use these classes together.

In some situations, performance improvements may be achievable, when you fall back to the detail level explained here.

8.3.1 Create a DisplayList

A *DisplayList* represents an interpreted document page. Methods for pixmap creation, text extraction and text search are – behind the curtain – all using the page’s display list to perform their tasks. If a page must be rendered several times (e.g. because of changed zoom levels), or if text search and text extraction should both be performed, overhead can be saved, if the display list is created only once and then used for all other tasks.

```
>>> dl = page.get_displaylist()           # create the display list
```

You can also create display lists for many pages “on stack” (in a list), may be during document open, during idling times, or you store it when a page is visited for the first time (e.g. in GUI scripts).

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

8.3.2 Generate Pixmap

The following creates a Pixmap from a *DisplayList*. Parameters are the same as for *Page.get_pixmap()*.

```
>>> pix = dl.get_pixmap()                 # create the page's pixmap
```

The execution time of this statement may be up to 50% shorter than that of *Page.get_pixmap()*.

8.3.3 Perform Text Search

With the display list from above, we can also search for text.

For this we need to create a *TextPage*.

```
>>> tp = dl.get_textpage()                 # display list from above
>>> rlist = tp.search("needle")            # look up "needle" locations
>>> for r in rlist:                        # work with the found locations, e.g.
    pix.invert_irect(r.irect)              # invert colors in the rectangles
```

8.3.4 Extract Text

With the same *TextPage* object from above, we can now immediately use any or all of the 5 text extraction methods.

Note: Above, we have created our text page without argument. This leads to a default argument of 3 (ligatures and white-space are preserved), IAW images will **not** be extracted – see below.

```
>>> txt = tp.extractText()           # plain text format
>>> json = tp.extractJSON()          # json format
>>> html = tp.extractHTML()          # HTML format
>>> xml = tp.extractXML()             # XML format
>>> xml = tp.extractXHTML()          # XHTML format
```

8.3.5 Further Performance improvements

8.3.5.1 Pixmap

As explained in the *Page* chapter:

If you do not need transparency set *alpha* = 0 when creating pixmaps. This will save 25% memory (if RGB, the most common case) and possibly 5% execution time (depending on the GUI software).

8.3.5.2 TextPage

If you do not need images extracted alongside the text of a page, you can set the following option:

```
>>> flags = fitz.TEXT_PRESERVE_LIGATURES | fitz.TEXT_PRESERVE_WHITESPACE
>>> tp = dl.get_textpage(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
```

GLOSSARY

matrix_like

A Python sequence of 6 numbers.

rect_like

A Python sequence of 4 numbers.

irect_like

A Python sequence of 4 integers.

point_like

A Python sequence of 2 numbers.

quad_like

A Python sequence of 4 *point_like* items.

inheritable

A number of values in a PDF can be inherited by objects further down in a parent-child relationship. The mediabox (physical size) of pages may for example be specified only once or in some node(s) of the *pagetree* and will then be taken as value for all *kids*, that do not specify their own value.

MediaBox

A PDF array of 4 floats specifying a physical page size – (*inheritable*, mandatory). This rectangle should contain all other PDF – optional – page rectangles, which may be specified in addition: CropBox, TrimBox, ArtBox and BleedBox. Please consult *Adobe PDF References* for details. The MediaBox is the only rectangle, for which there is no difference between MuPDF and PDF coordinate systems: *Page.mediabox* will always show the same coordinates as the */MediaBox* key in a page's object definition. For all other rectangles, MuPDF transforms coordinates such that the **top-left** corner is the point of reference. This can sometimes be confusing – you may for example encounter a situation like this one:

- The page definition contains the following identical values: */MediaBox* [36 45 607.5 765], */CropBox* [36 45 607.5 765].
- PyMuPDF accordingly shows *page.mediabox* = *Rect(36.0, 45.0, 607.5, 765.0)*.
- **BUT:** *page.cropbox* = *Rect(36.0, 0.0, 607.5, 720.0)*, because the two y-coordinates have been transformed (45 subtracted from both of them).

CropBox

A PDF array of 4 floats specifying a page's visible area – (*inheritable*, optional). It is the default for TrimBox, ArtBox and BleedBox. If not present, it defaults to MediaBox. This value is **not affected** if the page is rotated – in contrast to *Page.rect*. Also, other than the page rectangle, the top-left corner of the cropbox may or may not be (0, 0).

catalog

A central PDF *dictionary* – also called the “root” – containing document-wide parameters and pointers to many other information. Its *xref* is returned by *Document.pdf_catalog()*.

trailer

More precisely, the **PDF trailer** contains information in *dictionary* format. It is usually located at the file's end. In this dictionary, you will find things like the xrefs of the catalog and the metadata, the number of *xref* numbers, etc. Here is the definition of the PDF spec:

“The trailer of a PDF file enables an application reading the file to quickly find the cross-reference table and certain special objects. Applications should read a PDF file from its end.”

To access the trailer in PyMuPDF, use the usual methods `Document.xref_object()`, `Document.xref_get_key()` and `Document.xref_get_keys()` with -1 instead of a positive xref number.

contents

A **content stream** is a PDF *object* with an attached *stream*, whose data consists of a sequence of instructions describing the graphical elements to be painted on a page, see “Stream Objects” on page 19 of *Adobe PDF References*. For an overview of the mini-language used in these streams, see chapter “Operator Summary” on page 643 of the *Adobe PDF References*. A PDF *page* can have none to many contents objects. If it has none, the page is empty (but still may show annotations). If it has several, they will be interpreted in sequence as if their instructions had been present in one such object (i.e. like in a concatenated string). It should be noted that there are more stream object types which use the same syntax: e.g. appearance dictionaries associated with annotations and Form XObjects.

PyMuPDF provides a number of methods to deal with contents of PDF pages:

- `Page.read_contents()` – reads and concatenates all page contents into one bytes object.
- `Page.clean_contents()` – a wrapper of a MuPDF function that reads, concatenates and syntax-cleans all page contents. After this, only one /Contents object will exist. In addition, page *resources* will have been synchronized with it such that it will contain exactly those images, fonts and other objects that the page actually references.
- `Page.get_contents()` – return a list of *xref* numbers of a page's *contents* objects. May be empty. Use `Document.xref_stream()` with one of these xrefs to read the resp. contents section.
- `Page.set_contents()` – set a page's /Contents key to the provided *xref* number.

resources

A *dictionary* containing references to any resources (like images or fonts) required by a PDF *page* (required, inheritable, *Adobe PDF References* p. 81) and certain other objects (Form XObjects). This dictionary appears as a sub-dictionary in the object definition under the key `/Resources`. Being an inheritable object type, there may exist “parent” resources for all pages or certain subsets of pages.

dictionary

A PDF *object* type, which is somewhat comparable to the same-named Python notion: “A dictionary object is an associative table containing pairs of objects, known as the dictionary's entries. The first element of each entry is the key and the second element is the value. The key must be a name (...). The value can be any kind of object, including another dictionary. A dictionary entry whose value is null (...) is equivalent to an absent entry.” (*Adobe PDF References* p. 18).

Dictionaries are the most important *object* type in PDF. Here is an example (describing a *page*):

```
<<
/Contents 40 0 R           % value: an indirect object
/Type/Page                 % value: a name object
/MediaBox[0 0 595.32 841.92] % value: an array object
/Rotate 0                  % value: a number object
/Parent 12 0 R             % value: an indirect object
/Resources<<               % value: a dictionary object
  /ExtGState<</R7 26 0 R>>
  /Font<<
    /R8 27 0 R/R10 21 0 R/R12 24 0 R/R14 15 0 R
```

(continues on next page)

(continued from previous page)

```

    /R17 4 0 R/R20 30 0 R/R23 7 0 R /R27 20 0 R
  >>
  /ProcSet[/PDF/Text]           % value: array of two name objects
  >>
  /Annots[55 0 R]              % value: array, one entry (indirect object)
  >>

```

Contents, *Type*, *MediaBox*, etc. are **keys**, *40 0 R*, *Page*, *[0 0 595.32 841.92]*, etc. are the respective **values**. The strings “<<” and “>>” are used to enclose object definitions.

This example also shows the syntax of **nested** dictionary values: *Resources* has an object as its value, which in turn is a dictionary with keys like *ExtGState* (with the value <</R7 26 0 R>>, which is another dictionary), etc.

page

A PDF page is a *dictionary* object which defines one page in a PDF, see *Adobe PDF References* p. 71.

pagetree

The pages of a document are accessed through a structure known as the page tree, which defines the ordering of pages in the document. The tree structure allows PDF consumer applications, using only limited memory, to quickly open a document containing thousands of pages. The tree contains nodes of two types: intermediate nodes, called page tree nodes, and leaf nodes, called page objects. (*Adobe PDF References* p. 75).

While it is possible to list all page references in just one array, PDFs with many pages are often created using *balanced tree* structures (“page trees”) for faster access to any single page. In relation to the total number of pages, this can reduce the average page access time by page number from a linear to some logarithmic order of magnitude.

For fast page access, MuPDF can use its own array in memory – independently from what may or may not be present in the document file. This array is indexed by page number and therefore much faster than even the access via a perfectly balanced page tree.

object

Similar to Python, PDF supports the notion *object*, which can come in eight basic types: boolean values (“true” or “false”), integer and real numbers, strings (**always** enclosed in brackets – either “()”, or “<>” to indicate hexadecimal), names (must always start with a “/”, e.g. */Contents*), arrays (enclosed in brackets “[]”), dictionaries (enclosed in brackets “<<>>”), streams (enclosed by keywords “stream” / “endstream”), and the null object (“null”) (*Adobe PDF References* p. 13). Objects can be made identifiable by assigning a label. This label is then called *indirect* object. PyMuPDF supports retrieving definitions of indirect objects via their cross reference number via *Document.xref_object()*.

stream

A PDF *dictionary object* type which is followed by a sequence of bytes, similar to Python *bytes*. “However, a PDF application can read a stream incrementally, while a string must be read in its entirety. Furthermore, a stream can be of unlimited length, whereas a string is subject to an implementation limit. For this reason, objects with potentially large amounts of data, such as images and page descriptions, are represented as streams.” “A stream consists of a *dictionary* followed by zero or more bytes bracketed between the keywords *stream* and *endstream*”:

```

nnn 0 obj
<<
  dictionary definition
>>
stream
(zero or more bytes)

```

(continues on next page)

(continued from previous page)

```
endstream
endobj
```

See *Adobe PDF References* p. 19. PyMuPDF supports retrieving stream content via `Document.xref_stream()`. Use `Document.is_stream()` to determine whether an object is of stream type.

unitvector

A mathematical notion meaning a vector of norm (“length”) 1 – usually the Euclidean norm is implied. In PyMuPDF, this term is restricted to *Point* objects, see `Point.unit`.

xref

Abbreviation for cross-reference number: this is an integer unique identification for objects in a PDF. There exists a cross-reference table (which may physically consist of several separate segments) in each PDF, which stores the relative position of each object for quick lookup. The cross-reference table is one entry longer than the number of existing object: item zero is reserved and must not be used in any way. Many PyMuPDF classes have an *xref* attribute (which is zero for non-PDFs), and one can find out the total number of objects in a PDF via `Document.xref_length() - 1`.

resolution

Images and *Pixmap* objects may contain resolution information provided as “dots per inch”, dpi, in each direction (horizontal and vertical). When MuPDF reads an image from a file or from a PDF object, it will parse this information and put it in `Pixmap.xres`, `Pixmap.yres`, respectively. When it finds not meaningful information in the input (like non-positive values or values exceeding 4800), it will use “sane” defaults instead. The usual default value is 96, but it may also be 72 in some cases (e.g. for JPX images).

OCPD

Optional content properties dictionary - a sub *dictionary* of the PDF *catalog*. The central place to store optional content information, which is identified by the key `/OCProperties`. This dictionary has two required and one optional entry: (1) `/OCGs`, required, an array listing all optional content groups, (2) `/D`, required, the default optional content configuration dictionary (OCCD), (3) `/Configs`, optional, an array of alternative OCCDs.

OCCD

Optional content configuration dictionary - a PDF *dictionary* inside the PDF *OCPD*. It stores a setting of ON / OFF states of OCGs and how they are presented to a PDF viewer program. Selecting a configuration is quick way to achieve temporary mass visibility state changes. After opening a PDF, the `/D` configuration of the *OCPD* is always activated. Viewer should offer a way to switch between the `/D`, or one of the optional configurations contained in array `/Configs`.

OCG

Optional content group – a *dictionary* object used to control the visibility of other PDF objects like images or annotations. Independently on which page they are defined, objects with the same OCG can simultaneously be shown or hidden by setting their OCG to ON or OFF. This can be achieved via the user interface provided by many PDF viewers (Adobe Acrobat), or programmatically.

OCMD

Optional content membership dictionary – a *dictionary* object which can be used like an *OCG*: it has a visibility state. The visibility of an OCMD is **computed**: it is a logical expression, which uses the state of one or more OCGs to produce a boolean value. The expression’s result is interpreted as ON (true) or OFF (false).

ligature

Some frequent character combinations are represented by their own special glyphs in more advanced fonts. Typical examples are “fi”, “fl”, “ffi” and “ffl”. These compounds are called *ligatures*. In PyMuPDF text extractions, there is the option to either return the corresponding unicode unchanged, or split ligatures up into their constituent parts: “fi” ==> “f” + “i”, etc.

CONSTANTS AND ENUMERATIONS

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

10.1 Constants

Base14_Fonts

Predefined Python list of valid *PDF Base 14 Fonts*.

Return type

list

csRGB

Predefined RGB colorspace *fitz.Colorspace(fitz.CS_RGB)*.

Return type

Colorspace

csGRAY

Predefined GRAY colorspace *fitz.Colorspace(fitz.CS_GRAY)*.

Return type

Colorspace

csCMYK

Predefined CMYK colorspace *fitz.Colorspace(fitz.CS_CMYK)*.

Return type

Colorspace

CS_RGB

1 – Type of *Colorspace* is RGBA

Return type

int

CS_GRAY

2 – Type of *Colorspace* is GRAY

Return type

int

CS_CMYK

3 – Type of *Colorspace* is CMYK

Return type

int

VersionBind

'x.xx.x' – version of PyMuPDF (these bindings)

Return type
string

VersionFitz

'x.xxx' – version of MuPDF

Return type
string

VersionDate

ISO timestamp *YYYY-MM-DD HH:MM:SS* when these bindings were built.

Return type
string

Note: The docstring of *fitz* contains information of the above which can be retrieved like so: *print(fitz.__doc__)*, and should look like: *PyMuPDF 1.10.0: Python bindings for the MuPDF 1.10 library, built on 2016-11-30 13:09:13.*

version

(VersionBind, VersionFitz, timestamp) – combined version information where *timestamp* is the generation point in time formatted as “YYYYMMDDhhmmss”.

Return type
tuple

10.2 Document Permissions

Code	Permitted Action
PDF_PERM_PRINT	Print the document
PDF_PERM_MODIFY	Modify the document's contents
PDF_PERM_COPY	Copy or otherwise extract text and graphics
PDF_PERM_ANNOTATE	Add or modify text annotations and interactive form fields
PDF_PERM_FORM	Fill in forms and sign the document
PDF_PERM_ACCESSIBILITY	Obsolete, always permitted
PDF_PERM_ASSEMBLE	Insert, rotate, or delete pages, bookmarks, thumbnail images
PDF_PERM_PRINT_HQ	High quality printing

10.3 PDF encryption method codes

Code	Meaning
PDF_ENCRYPT_KEEP	do not change
PDF_ENCRYPT_NONE	remove any encryption
PDF_ENCRYPT_RC4_40	RC4 40 bit
PDF_ENCRYPT_RC4_128	RC4 128 bit
PDF_ENCRYPT_AES_128	<i>Advanced Encryption Standard 128 bit</i>
PDF_ENCRYPT_AES_256	<i>Advanced Encryption Standard 256 bit</i>
PDF_ENCRYPT_UNKNOWN	unknown

10.4 Font File Extensions

The table show file extensions you should use when saving fontfile buffers extracted from a PDF. This string is returned by `Document.get_page_fonts()`, `Page.get_fonts()` and `Document.extract_font()`.

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	not extractable, e.g. <i>PDF Base 14 Fonts</i> , Type 3 fonts and others

10.5 Text Alignment

TEXT_ALIGN_LEFT

0 – align left.

TEXT_ALIGN_CENTER

1 – align center.

TEXT_ALIGN_RIGHT

2 – align right.

TEXT_ALIGN_JUSTIFY

3 – align justify.

10.6 Text Extraction Flags

Option bits controlling the amount of data, that are parsed into a *TextPage* – this class is mainly used only internally in PyMuPDF.

For the PyMuPDF programmer, some combination (using Python’s `|` operator, or simply use `+`) of these values are aggregated in the `flags` integer, a parameter of all text search and text extraction methods. Depending on the individual method, different default combinations of the values are used. Please use a value that meets your situation. Especially make sure to switch off image extraction unless you really need them. The impact on performance and memory is significant!

TEXT_PRESERVE_LIGATURES

1 – If set, ligatures are passed through to the application in their original form. Otherwise ligatures are expanded into their constituent parts, e.g. the ligature “ffi” is expanded into three eparate characters f, f and i. Default is “on” in PyMuPDF. MuPDF supports the following 7 ligatures: “ff”, “fi”, “fl”, “ffi”, “ffl”, “ft”, “st”.

TEXT_PRESERVE_WHITESPACE

2 – If set, whitespace is passed through. Otherwise any type of horizontal whitespace (including horizontal tabs) will be replaced with space characters of variable width. Default is “on” in PyMuPDF.

TEXT_PRESERVE_IMAGES

4 – If set, then images will be stored in the *TextPage*. This causes the presence of (usually large!) binary image content in the output of text extractions of types “blocks”, “dict”, “json”, “rawdict”, “rawjson”, “html”, and

“xhtml” and is the default there. If used with “blocks” however, only image metadata will be returned, not the image itself.

TEXT_INHIBIT_SPACES

8 – If set, Mupdf will not try to add missing space characters where there are large gaps between characters. In PDF, the creator often does not insert spaces to point to the next character’s position, but will provide the direct location address. The default in PyMuPDF is “off” – so spaces **will be generated**.

TEXT_DEHYPHENATE

16 – Ignore hyphens at line ends and join with next line. Used internally with the text search functions. However, it is generally available: if on, text extractions will return joined text lines (or spans) with the ending hyphen of the first line eliminated. So two separate spans “**first meth-**” and “**od leads to wrong results**” on different lines will be joined to one span “**first method leads to wrong results**” and correspondingly updated bboxes: the characters of the resulting span will no longer have identical y-coordinates.

TEXT_PRESERVE_SPANS

32 – Generate a new line for every span. Not used (“off”) in PyMuPDF, but available for your use. Every line in “dict”, “json”, “rawdict”, “rawjson” will contain exactly one span.

TEXT_MEDIABOX_CLIP

64 – If set, characters entirely outside a page’s **mediabox** will be ignored. This is default in PyMuPDF.

The following constants represent the default combinations of the above for text extraction and searching:

TEXTFLAGS_TEXT

TEXTFLAGS_WORDS

TEXTFLAGS_BLOCKS

TEXTFLAGS_DICT

TEXTFLAGS_RAWDICT

TEXTFLAGS_HTML

TEXTFLAGS_XHTML

TEXTFLAGS_XML

TEXTFLAGS_SEARCH

10.7 Link Destination Kinds

Possible values of *linkDest.kind* (link destination kind).

LINK_NONE

0 – No destination. Indicates a dummy link.

Return type

int

LINK_GOTO

1 – Points to a place in this document.

Return type

int

LINK_URI

2 – Points to a URI – typically a resource specified with internet syntax.

Return type

int

LINK_LAUNCH

3 – Launch (open) another file (of any “executable” type).

Return type

int

LINK_NAMED

4 – points to a named location.

Return type

int

LINK_GOTOR

5 – Points to a place in another PDF document.

Return type

int

10.8 Link Destination Flags

Note: The rightmost byte of this integer is a bit field, so test the truth of these bits with the & operator.

LINK_FLAG_L_VALID

1 (bit 0) Top left x value is valid

Return type

bool

LINK_FLAG_T_VALID

2 (bit 1) Top left y value is valid

Return type

bool

LINK_FLAG_R_VALID

4 (bit 2) Bottom right x value is valid

Return type

bool

LINK_FLAG_B_VALID

8 (bit 3) Bottom right y value is valid

Return type

bool

LINK_FLAG_FIT_H

16 (bit 4) Horizontal fit

Return type

bool

LINK_FLAG_FIT_V

32 (bit 5) Vertical fit

Return type

bool

LINK_FLAG_R_IS_ZOOM

64 (bit 6) Bottom right x is a zoom figure

Return type

bool

10.9 Annotation Related Constants

See chapter 8.4.5, pp. 615 of the *Adobe PDF References* for details.

10.9.1 Annotation Types

These identifiers also cover **links** and **widgets**: the PDF specification technically handles them all in the same way, whereas **MuPDF** (and PyMuPDF) treats them as three basically different types of objects.

```
PDF_ANNOT_TEXT 0
PDF_ANNOT_LINK 1 # <=== Link object in PyMuPDF
PDF_ANNOT_FREE_TEXT 2
PDF_ANNOT_LINE 3
PDF_ANNOT_SQUARE 4
PDF_ANNOT_CIRCLE 5
PDF_ANNOT_POLYGON 6
PDF_ANNOT_POLY_LINE 7
PDF_ANNOT_HIGHLIGHT 8
PDF_ANNOT_UNDERLINE 9
PDF_ANNOT_SQUIGGLY 10
PDF_ANNOT_STRIKE_OUT 11
PDF_ANNOT_REDACT 12
PDF_ANNOT_STAMP 13
PDF_ANNOT_CARET 14
PDF_ANNOT_INK 15
PDF_ANNOT_POPUP 16
PDF_ANNOT_FILE_ATTACHMENT 17
PDF_ANNOT_SOUND 18
PDF_ANNOT_MOVIE 19
PDF_ANNOT_RICH_MEDIA 20
PDF_ANNOT_WIDGET 21 # <=== Widget object in PyMuPDF
PDF_ANNOT_SCREEN 22
PDF_ANNOT_PRINTER_MARK 23
PDF_ANNOT_TRAP_NET 24
PDF_ANNOT_WATERMARK 25
PDF_ANNOT_3D 26
PDF_ANNOT_PROJECTION 27
PDF_ANNOT_UNKNOWN -1
```


10.9.2 Annotation Flag Bits

```
PDF_ANNOT_IS_INVISIBLE 1 << (1-1)
PDF_ANNOT_IS_HIDDEN 1 << (2-1)
PDF_ANNOT_IS_PRINT 1 << (3-1)
PDF_ANNOT_IS_NO_ZOOM 1 << (4-1)
PDF_ANNOT_IS_NO_ROTATE 1 << (5-1)
PDF_ANNOT_IS_NO_VIEW 1 << (6-1)
PDF_ANNOT_IS_READ_ONLY 1 << (7-1)
PDF_ANNOT_IS_LOCKED 1 << (8-1)
PDF_ANNOT_IS_TOGGLE_NO_VIEW 1 << (9-1)
PDF_ANNOT_IS_LOCKED_CONTENTS 1 << (10-1)
```

10.9.3 Annotation Line Ending Styles

```
PDF_ANNOT_LE_NONE 0
PDF_ANNOT_LE_SQUARE 1
PDF_ANNOT_LE_CIRCLE 2
PDF_ANNOT_LE_DIAMOND 3
PDF_ANNOT_LE_OPEN_ARROW 4
PDF_ANNOT_LE_CLOSED_ARROW 5
PDF_ANNOT_LE_BUTT 6
PDF_ANNOT_LE_R_OPEN_ARROW 7
PDF_ANNOT_LE_R_CLOSED_ARROW 8
PDF_ANNOT_LE_SLASH 9
```

10.10 Widget Constants

10.10.1 Widget Types (*field_type*)

```
PDF_WIDGET_TYPE_UNKNOWN 0
PDF_WIDGET_TYPE_BUTTON 1
PDF_WIDGET_TYPE_CHECKBOX 2
PDF_WIDGET_TYPE_COMBOBOX 3
PDF_WIDGET_TYPE_LISTBOX 4
PDF_WIDGET_TYPE_RADIOBUTTON 5
PDF_WIDGET_TYPE_SIGNATURE 6
PDF_WIDGET_TYPE_TEXT 7
```

10.10.2 Text Widget Subtypes (*text_format*)

```
PDF_WIDGET_TX_FORMAT_NONE 0
PDF_WIDGET_TX_FORMAT_NUMBER 1
PDF_WIDGET_TX_FORMAT_SPECIAL 2
PDF_WIDGET_TX_FORMAT_DATE 3
PDF_WIDGET_TX_FORMAT_TIME 4
```

10.10.3 Widget flags (*field_flags*)

Common to all field types:

```
PDF_FIELD_IS_READ_ONLY 1
PDF_FIELD_IS_REQUIRED 1 << 1
PDF_FIELD_IS_NO_EXPORT 1 << 2
```

Text widgets:

```
PDF_TX_FIELD_IS_MULTILINE 1 << 12
PDF_TX_FIELD_IS_PASSWORD 1 << 13
PDF_TX_FIELD_IS_FILE_SELECT 1 << 20
PDF_TX_FIELD_IS_DO_NOT_SPELL_CHECK 1 << 22
PDF_TX_FIELD_IS_DO_NOT_SCROLL 1 << 23
PDF_TX_FIELD_IS_COMB 1 << 24
PDF_TX_FIELD_IS_RICH_TEXT 1 << 25
```

Button widgets:

```
PDF_BTN_FIELD_IS_NO_TOGGLE_TO_OFF 1 << 14
PDF_BTN_FIELD_IS_RADIO 1 << 15
PDF_BTN_FIELD_IS_PUSHBUTTON 1 << 16
PDF_BTN_FIELD_IS_RADIOS_IN_UNISON 1 << 25
```

Choice widgets:

```
PDF_CH_FIELD_IS_COMBO 1 << 17
PDF_CH_FIELD_IS_EDIT 1 << 18
PDF_CH_FIELD_IS_SORT 1 << 19
PDF_CH_FIELD_IS_MULTI_SELECT 1 << 21
PDF_CH_FIELD_IS_DO_NOT_SPELL_CHECK 1 << 22
PDF_CH_FIELD_IS_COMMIT_ON_SEL_CHANGE 1 << 26
```

10.11 PDF Standard Blend Modes

For an explanation see *Adobe PDF References*, page 324:

```
PDF_BM_Color "Color"
PDF_BM_ColorBurn "ColorBurn"
PDF_BM_ColorDodge "ColorDodge"
PDF_BM_Darken "Darken"
```

(continues on next page)

(continued from previous page)

```

PDF_BM_Difference "Difference"
PDF_BM_Exclusion  "Exclusion"
PDF_BM_HardLight  "HardLight"
PDF_BM_Hue        "Hue"
PDF_BM_Lighten    "Lighten"
PDF_BM_Luminosity "Luminosity"
PDF_BM_Multiply    "Multiply"
PDF_BM_Normal      "Normal"
PDF_BM_Overlay     "Overlay"
PDF_BM_Saturation  "Saturation"
PDF_BM_Screen      "Screen"
PDF_BM_SoftLight   "Softlight"

```

10.12 Stamp Annotation Icons

MuPDF has defined the following icons for **rubber stamp** annotations:

```

STAMP_Approved 0
STAMP_AsIs     1
STAMP_Confidential 2
STAMP_Departmental 3
STAMP_Experimental 4
STAMP_Expired   5
STAMP_Final     6
STAMP_ForComment 7
STAMP_ForPublicRelease 8
STAMP_NotApproved 9
STAMP_NotForPublicRelease 10
STAMP_Sold      11
STAMP_TopSecret 12
STAMP_Draft     13

```


COLOR DATABASE

Since the introduction of methods involving colors (like `Page.draw_circle()`), a requirement may be to have access to predefined colors.

The fabulous GUI package `wxPython` has a database of over 540 predefined RGB colors, which are given more or less memorable names. Among them are not only standard names like “green” or “blue”, but also “turquoise”, “skyblue”, and 100 (not only 50 ...) shades of “gray”, etc.

We have taken the liberty to copy this database (a list of tuples) modified into PyMuPDF and make its colors available as PDF compatible float triples: for wxPython’s (“WHITE”, 255, 255, 255) we return (1, 1, 1), which can be directly used in *color* and *fill* parameters. We also accept any mixed case of “wHiTe” to find a color.

11.1 Function `getColor()`

As the color database may not be needed very often, one additional import statement seems acceptable to get access to it:

```
>>> # "getColor" is the only method you really need
>>> from fitz.utils import getColor
>>> getColor("aliceblue")
(0.9411764705882353, 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)] ...
```

11.2 Printing the Color Database

If you want to actually see how the many available colors look like, use scripts [colordbRGB.py](#) or [colordbHSV.py](#) in the examples directory. They create PDFs (already existing in the same directory) with all these colors. Their only difference is sorting order: one takes the RGB values, the other one the Hue-Saturation-Values as sort criteria. This is a screen print of what these files look like.



APPENDIX 1: DETAILS ON TEXT EXTRACTION

This chapter provides background on the text extraction methods of PyMuPDF.

Information of interest are

- what do they provide?
- what do they imply (processing time / data sizes)?

12.1 General structure of a *TextPage*

TextPage is one of (Py-) MuPDF's classes. It is normally created (and destroyed again) behind the curtain, when *Page* text extraction methods are used, but it is also available directly and can be used as a persistent object. Other than its name suggests, images may optionally also be part of a text page:

```
<page>
  <text block>
    <line>
      <span>
        <char>
      </span>
    </line>
  </text block>
  <image block>
    <img>
  </image block>
</page>
```

A **text page** consists of blocks (= roughly paragraphs).

A **block** consists of either lines and their characters, or an image.

A **line** consists of spans.

A **span** consists of adjacent characters with identical font properties: name, size, flags and color.

12.2 Plain Text

Function *TextPage.extractText()* (or *Page.get_text("text")*) extracts a page's plain **text in original order** as specified by the creator of the document.

An example output:

```
>>> print(page.get_text("text"))
Some text on first page.
```

Note: The output may not equal an accustomed “natural” reading order. However, you can request a reordering following the scheme “top-left to bottom-right” by executing `page.get_text(“text”, sort=True)`.

12.3 BLOCKS

Function `TextPage.extractBLOCKS()` (or `Page.get_text(“blocks”)`) extracts a page’s text blocks as a list of items like:

```
(x0, y0, x1, y1, "lines in block", block_no, block_type)
```

Where the first 4 items are the float coordinates of the block’s bbox. The lines within each block are concatenated by a new-line character.

This is a high-speed method, which by default also extracts image meta information: Each image appears as a block with one text line, which contains meta information. The image itself is not shown.

As with simple text output above, the `sort` argument can be used as well to obtain a reading order.

Example output:

```
>>> print(page.get_text("blocks", sort=False))
[(50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375,
'Some text on first page.', 0, 0)]
```

12.4 WORDS

Function `TextPage.extractWORDS()` (or `Page.get_text(“words”)`) extracts a page’s text **words** as a list of items like:

```
(x0, y0, x1, y1, "word", block_no, line_no, word_no)
```

Where the first 4 items are the float coordinates of the words’s bbox. The last three integers provide some more information on the word’s whereabouts.

This is a high-speed method. As with the previous methods, argument `sort=True` will reorder the words.

Example output:

```
>>> for word in page.get_text("words", sort=False):
    print(word)
(50.0, 88.17500305175781, 78.73200225830078, 103.28900146484375,
'Some', 0, 0, 0)
(81.79000091552734, 88.17500305175781, 99.5219955444336, 103.28900146484375,
'text', 0, 0, 1)
(102.57999420166016, 88.17500305175781, 114.8119888305664, 103.28900146484375,
'on', 0, 0, 2)
(117.86998748779297, 88.17500305175781, 135.5909881591797, 103.28900146484375,
'first', 0, 0, 3)
(138.64898681640625, 88.17500305175781, 166.1709747314453, 103.28900146484375,
'page.', 0, 0, 4)
```


12.5 HTML

`TextPage.extractHTML()` (or `Page.get_text("html")`) output fully reflects the structure of the page's *TextPage* – much like DICT / JSON below. This includes images, font information and text positions. If wrapped in HTML header and trailer code, it can readily be displayed by an internet browser. Our above example:

```
>>> for line in page.get_text("html").splitlines():
    print(line)

<div id="page0" style="position:relative;width:300pt;height:350pt;
background-color:white">
<p style="position:absolute;white-space:pre;margin:0;padding:0;top:88pt;
left:50pt"><span style="font-family:Helvetica,sans-serif;
font-size:11pt">Some text on first page.</span></p>
</div>
```

12.6 Controlling Quality of HTML Output

While HTML output has improved a lot in MuPDF v1.12.0, it is not yet bug-free: we have found problems in the areas **font support** and **image positioning**.

- HTML text contains references to the fonts used of the original document. If these are not known to the browser (a fat chance!), it will replace them with others; the results will probably look awkward. This issue varies greatly by browser – on my Windows machine, MS Edge worked just fine, whereas Firefox looked horrible.
- For PDFs with a complex structure, images may not be positioned and / or sized correctly. This seems to be the case for rotated pages and pages, where the various possible page bbox variants do not coincide (e.g. *MediaBox* != *CropBox*). We do not know yet, how to address this – we filed a bug at MuPDF's site.

To address the font issue, you can use a simple utility script to scan through the HTML file and replace font references. Here is a little example that replaces all fonts with one of the *PDF Base 14 Fonts*: serifed fonts will become “Times”, non-serifed “Helvetica” and monospaced will become “Courier”. Their respective variations for “bold”, “italic”, etc. are hopefully done correctly by your browser:

```
import sys
filename = sys.argv[1]
otext = open(filename).read()           # original html text string
pos1 = 0                                # search start poition
font_serif = "font-family:Times"        # enter ...
font_sans = "font-family:Helvetica"     # ... your choices ...
font_mono = "font-family:Courier"       # ... here
found_one = False                       # true if search successfull

while True:
    pos0 = otext.find("font-family:", pos1) # start of a font spec
    if pos0 < 0:                             # none found - we are done
        break
    pos1 = otext.find(";", pos0)             # end of font spec
    test = otext[pos0 : pos1]               # complete font spec string
    testn = ""                             # the new font spec string
    if test.endswith(",serif"):              # font with serifs?
        testn = font_serif                 # use Times instead
```

(continues on next page)

(continued from previous page)

```

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!")

```

12.7 DICT (or JSON)

`TextPage.extractDICT()` (or `Page.get_text("dict", sort=False)`) output fully reflects the structure of a *TextPage* and provides image content and position detail (*bbox* – boundary boxes in pixel units) for every block, line and span. Images are stored as *bytes* for DICT output and base64 encoded strings for JSON output.

For a visualization of the dictionary structure have a look at *Structure of Dictionary Outputs*.

Here is how this looks like:

```

{
  "width": 300.0,
  "height": 350.0,
  "blocks": [{
    "type": 0,
    "bbox": (50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375),
    "lines": ({
      "wmode": 0,
      "dir": (1.0, 0.0),
      "bbox": (50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375),
      "spans": ({
        "size": 11.0,
        "flags": 0,
        "font": "Helvetica",
        "color": 0,
        "origin": (50.0, 100.0),
        "text": "Some text on first page.",
        "bbox": (50.0, 88.17500305175781, 166.1709747314453, 103.28900146484375)
      })
    })
  ]
}

```

12.8 RAWDICT (or RAWJSON)

`TextPage.extractRAWDICT()` (or `Page.get_text("rawdict", sort=False)`) 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*) in the spans are replaced by the list *“chars”*. Each *“chars”* entry is a character *dict*. For example, here is what you would see in place of item *“text”*: *“Text in black color.”* above:

```
"chars": [{
  "origin": (50.0, 100.0),
  "bbox": (50.0, 88.17500305175781, 57.336997985839844, 103.28900146484375),
  "c": "S"
}, {
  "origin": (57.33700180053711, 100.0),
  "bbox": (57.33700180053711, 88.17500305175781, 63.4530029296875, 103.28900146484375),
  "c": "O"
}, {
  "origin": (63.4530029296875, 100.0),
  "bbox": (63.4530029296875, 88.17500305175781, 72.61600494384766, 103.28900146484375),
  "c": "m"
}, {
  "origin": (72.61600494384766, 100.0),
  "bbox": (72.61600494384766, 88.17500305175781, 78.73200225830078, 103.
↪28900146484375),
  "c": "e"
}, {
  "origin": (78.73200225830078, 100.0),
  "bbox": (78.73200225830078, 88.17500305175781, 81.79000091552734, 103.
↪28900146484375),
  "c": " "
< ... deleted ... >
}, {
  "origin": (163.11297607421875, 100.0),
  "bbox": (163.11297607421875, 88.17500305175781, 166.1709747314453, 103.
↪28900146484375),
  "c": "."
}],
```

12.9 XML

The `TextPage.extractXML()` (or `Page.get_text("xml")`) version extracts text (no images) with the detail level of RAWDICT:

```
>>> for line in page.get_text("xml").splitlines():
      print(line)

<page id="page0" width="300" height="350">
<block bbox="50 88.175 166.17098 103.289">
<line bbox="50 88.175 166.17098 103.289" wmode="0" dir="1 0">
<font name="Helvetica" size="11">
<char quad="50 88.175 57.336999 88.175 50 103.289 57.336999 103.289" x="50"
y="100" color="#000000" c="S"/>
```

(continues on next page)

(continued from previous page)

```

<char quad="57.337 88.175 63.453004 88.175 57.337 103.289 63.453004 103.289" x="57.337"
y="100" color="#000000" c="o"/>
<char quad="63.453004 88.175 72.616008 88.175 63.453004 103.289 72.616008 103.289" x="63.
↪ 453004"
y="100" color="#000000" c="m"/>
<char quad="72.616008 88.175 78.732 88.175 72.616008 103.289 78.732 103.289" x="72.616008
↪ "
y="100" color="#000000" c="e"/>
<char quad="78.732 88.175 81.79 88.175 78.732 103.289 81.79 103.289" x="78.732"
y="100" color="#000000" c=" " />

... deleted ...

<char quad="163.11298 88.175 166.17098 88.175 163.11298 103.289 166.17098 103.289" x=
↪ "163.11298"
y="100" color="#000000" c="."/>
</font>
</line>
</block>
</page>

```

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

12.10 XHTML

`TextPage.extractXHTML()` (or `Page.get_text("xhtml")`) is a variation of TEXT but in HTML format, containing the bare text and images (“semantic” output):

```

<div id="page0">
<p>Some text on first page.</p>
</div>

```

12.11 Text Extraction Flags Defaults

- New in version 1.16.2: Method `Page.get_text()` supports a keyword parameter `flags (int)` to control the amount and the quality of extracted data. The following table shows the defaults settings (flags parameter omitted or None) for each extraction variant. If you specify flags with a value other than *None*, be aware that you must set **all desired** options. A description of the respective bit settings can be found in *Text Extraction Flags*.
- New in v1.19.6: The default combinations in the following table are now available as Python constants: `TEXTFLAGS_TEXT`, `TEXTFLAGS_WORDS`, `TEXTFLAGS_BLOCKS`, `TEXTFLAGS_DICT`, `TEXTFLAGS_RAWDICT`, `TEXTFLAGS_HTML`, `TEXTFLAGS_XHTML`, `TEXTFLAGS_XML`, and `TEXTFLAGS_SEARCH`. You can now easily modify a default flag, e.g.

– **include** images in a “blocks” output:

```
flags = TEXTFLAGS_BLOCKS | TEXT_PRESERVE_IMAGES
```

– **exclude** images from a “dict” output:

```
flags = TEXTFLAGS_DICT & ~TEXT_PRESERVE_IMAGES
```

– set **dehyphenation off** in text searches:

```
flags = TEXTFLAGS_SEARCH & ~TEXT_DEHYPHENATE
```

Indicator	text	html	xhtml	xml	dict	rawdict	words	blocks	search
preserve ligatures	1	1	1	1	1	1	1	1	1
preserve whitespace	1	1	1	1	1	1	1	1	1
preserve images	n/a	1	1	n/a	1	1	n/a	0	0
inhibit spaces	0	0	0	0	0	0	0	0	0
dehyphenate	0	0	0	0	0	0	0	0	1
clip to mediabox	1	1	1	1	1	1	1	1	1

- **search** refers to the text search function.
- “**json**” is handled exactly like “**dict**” and is hence left out.
- “**rawjson**” is handled exactly like “**rawdict**” and is hence left out.
- An “n/a” specification means a value of 0 and setting this bit never has any effect on the output (but an adverse effect on performance).
- If you are not interested in images when using an output variant which includes them by default, then by all means set the respective bit off: You will experience a better performance and much lower space requirements.

To show the effect of `TEXT_INHIBIT_SPACES` have a look at this example:

```
>>> print(page.get_text("text"))
H a l l o !
M o r e   t e x t
i s   f o l l o w i n g
i n   E n g l i s h
. . .   l e t ' s   s e e
w h a t   h a p p e n s .
>>> print(page.get_text("text", flags=fitz.TEXT_INHIBIT_SPACES))
Hallo!
More text
is following
in English
... let's see
what happens.
>>>
```

12.12 Performance

The text extraction methods differ significantly both: in terms of information they supply, and in terms of resource requirements and runtimes. Generally, more information of course means, that more processing is required and a higher data volume is generated.

Note: Especially images have a **very significant** impact. Make sure to exclude them (via the *flags* parameter) whenever you do not need them. To process the below mentioned 2’700 total pages with default flags settings required 160 seconds across all extraction methods. When all images were excluded, less than 50% of that time (77 seconds) were needed.

To begin with, all methods are **very fast** in relation to other products out there in the market. In terms of processing speed, we are not aware of a faster (free) tool. Even the most detailed method, RAWDICT, processes all 1'310 pages of the *Adobe PDF References* in less than 5 seconds (simple text needs less than 2 seconds here).

The following table shows average relative speeds ("RSpeed", baseline 1.00 is TEXT), taken across ca. 1400 text-heavy and 1300 image-heavy pages.

Method	RSpeed	Comments	no images
TEXT	1.00	no images, plain text, line breaks	1.00
BLOCKS	1.00	image bboxes (only), block level text with bboxes, line breaks	1.00
WORDS	1.02	no images, word level text with bboxes	1.02
XML	2.72	no images, char level text, layout and font details	2.72
XHTML	3.32	base64 images, span level text, no layout info	1.00
HTML	3.54	base64 images, span level text, layout and font details	1.01
DICT	3.93	binary images, span level text, layout and font details	1.04
RAWDICT	4.50	binary images, char level text, layout and font details	1.68

As mentioned: when excluding image extraction (last column), the relative speeds are changing drastically: except RAWDICT and XML, the other methods are almost equally fast, and RAWDICT requires 40% less execution time than the **now slowest XML**.

Look at chapter **Appendix 1** for more performance information.

APPENDIX 2: CONSIDERATIONS ON EMBEDDED FILES

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

13.1 General

Starting with version 1.4, PDF supports embedding arbitrary files as part (“Embedded File Streams”) of a PDF document file (see chapter “7.11.4 Embedded File Streams”, pp. 103 of the *Adobe PDF References*).

In many aspects, this is comparable to concepts also found in ZIP files or the OLE technique in MS Windows. PDF embedded files do, however, *not* support directory structures as does the ZIP format. An embedded file can in turn contain embedded files itself.

Advantages of this concept are that embedded files are under the PDF umbrella, benefitting from its permissions / password protection and integrity aspects: all data, which a PDF may reference or even may be dependent on, can be bundled into it and so form a single, consistent unit of information.

In addition to embedded files, PDF 1.7 adds *collections* to its support range. This is an advanced way of storing and presenting meta information (i.e. arbitrary and extensible properties) of embedded files.

13.2 MuPDF Support

After adding initial support for collections (portfolios) and */EmbeddedFiles* in MuPDF version 1.11, this support was dropped again in version 1.15.

As a consequence, the cli utility *mutool* no longer offers access to embedded files.

PyMuPDF – having implemented an */EmbeddedFiles* API in response in its version 1.11.0 – was therefore forced to change gears starting with its version 1.16.0 (we never published a MuPDF v1.15.x compatible PyMuPDF).

We are now maintaining our own code basis supporting embedded files. This code makes use of basic MuPDF dictionary and array functions only.

13.3 PyMuPDF Support

We continue to support the full old API with respect to embedded files – with only minor, cosmetic changes.

There even also is a new function, which delivers a list of all names under which embedded data are resgistered in a PDF, `Document.embfile_names()`.

APPENDIX 3: ASSORTED TECHNICAL INFORMATION

This section deals with various technical topics, that are not necessarily related to each other.

14.1 Image Transformation Matrix

Starting with version 1.18.11, the image transformation matrix is returned by some methods for text and image extraction: `Page.get_text()` and `Page.get_image_bbox()`.

The transformation matrix contains information about how an image was transformed to fit into the rectangle (its “boundary box” = “bbox”) on some document page. By inspecting the image’s bbox on the page and this matrix, one can determine for example, whether and how the image is displayed scaled or rotated on a page.

The relationship between image dimension and its bbox on a page is the following:

1. Using the original image’s width and height,

- define the image rectangle `imgrect = fitz.Rect(0, 0, width, height)`
- define the “shrink matrix” `shrink = fitz.Matrix(1/width, 0, 0, 1/height, 0, 0)`.

2. Transforming the image rectangle with its shrink matrix, will result in the unit rectangle: `imgrect * shrink = fitz.Rect(0, 0, 1, 1)`.

3. Using the image **transformation matrix** “transform”, the following steps will compute the bbox:

```
imgrect = fitz.Rect(0, 0, width, height)
shrink = fitz.Matrix(1/width, 0, 0, 1/height, 0, 0)
bbox = imgrect * shrink * transform
```

4. Inspecting the matrix product `shrink * transform` will reveal all information about what happened to the image rectangle to make it fit into the bbox on the page: rotation, scaling of its sides and translation of its origin. Let us look at an example:

```
>>> imginfo = page.get_images()[0] # get an image item on a page
>>> imginfo
(5, 0, 439, 501, 8, 'DeviceRGB', '', 'fzImg0', 'DCTDecode')
>>> #-----
>>> # define image shrink matrix and rectangle
>>> #-----
>>> shrink = fitz.Matrix(1 / 439, 0, 0, 1 / 501, 0, 0)
>>> imgrect = fitz.Rect(0, 0, 439, 501)
>>> #-----
```

(continues on next page)

(continued from previous page)

```

>>> # determine image bbox and transformation matrix:
>>> #-----
>>> bbox, transform = page.get_image_bbox("fzImg0", transform=True)
>>> #-----
>>> # confirm equality - permitting rounding errors
>>> #-----
>>> bbox
Rect(100.0, 112.37525939941406, 300.0, 287.624755859375)
>>> imgrect * shrink * transform
Rect(100.0, 112.375244140625, 300.0, 287.6247253417969)
>>> #-----
>>> shrink * transform
Matrix(0.0, -0.39920157194137573, 0.3992016017436981, 0.0, 100.0, 287.6247253417969)
>>> #-----
>>> # the above shows:
>>> # image sides are scaled by same factor ~0.4,
>>> # and the image is rotated by 90 degrees clockwise
>>> # compare this with fitz.Matrix(-90) * 0.4
>>> #-----

```

14.2 PDF Base 14 Fonts

The following 14 builtin font names **must be supported by every PDF viewer** application. They are available as a dictionary, which maps their full names and their abbreviations in lower case to the full font basename. Wherever a **fontname** must be provided in PyMuPDF, any **key or value** from the dictionary may be used:

```

In [2]: fitz.Base14_fontdict
Out[2]:
{'courier': 'Courier',
'courier-oblique': 'Courier-Oblique',
'courier-bold': 'Courier-Bold',
'courier-boldoblique': 'Courier-BoldOblique',
'helvetica': 'Helvetica',
'helvetica-oblique': 'Helvetica-Oblique',
'helvetica-bold': 'Helvetica-Bold',
'helvetica-boldoblique': 'Helvetica-BoldOblique',
'times-roman': 'Times-Roman',
'times-italic': 'Times-Italic',
'times-bold': 'Times-Bold',
'times-bolditalic': 'Times-BoldItalic',
'symbol': 'Symbol',
'zapfdingbats': 'ZapfDingbats',
'helv': 'Helvetica',
'heit': 'Helvetica-Oblique',
'hebo': 'Helvetica-Bold',
'hebi': 'Helvetica-BoldOblique',
'cour': 'Courier',
'coit': 'Courier-Oblique',
'cobo': 'Courier-Bold',

```

(continues on next page)

(continued from previous page)

```
'cobi': 'Courier-BoldOblique',
'tiro': 'Times-Roman',
'tibo': 'Times-Bold',
'tiit': 'Times-Italic',
'tibi': 'Times-BoldItalic',
'symb': 'Symbol',
'zadb': 'ZapfDingbats'}
```

In contrast to their obligation, not all PDF viewers support these fonts correctly and completely – this is especially true for Symbol and ZapfDingbats. Also, the glyph (visual) images will be specific to every reader.

To see how these fonts can be used – including the **CJK built-in** fonts – look at the table in [Page.insert_font\(\)](#).

14.3 Adobe PDF References

This PDF Reference manual published by Adobe is frequently quoted throughout this documentation. It can be viewed and downloaded from [here](#).

Note: For a long time, an older version was also available under [this](#) link. It seems to be taken off of the web site in October 2021. Earlier (pre 1.19.*) versions of the PyMuPDF documentation used to refer to this document. We have undertaken an effort to replace referrals to the current specification above.

14.4 Using Python Sequences as Arguments in PyMuPDF

When PyMuPDF objects and methods require a Python **list** of numerical values, other Python **sequence types** are also allowed. Python classes are said to implement the **sequence protocol**, if they have a `__getitem__()` method.

This basically means, you can interchangeably use Python *list* or *tuple* or even *array.array*, *numpy.array* and *bytearray* types in these cases.

For example, specifying a sequence "s" in any of the following ways

- `s = [1, 2]` – a list
- `s = (1, 2)` – a tuple
- `s = array.array("i", (1, 2))` – an `array.array`
- `s = numpy.array((1, 2))` – a `numpy.array`
- `s = bytearray((1, 2))` – a `bytearray`

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.

14.5 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()*, *delete_page()*, *insert_page()* ... 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()*, *insert_page()*, *insert_pdf()* 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.first_annot
>>> 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].first_annot                # 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.first_annot
>>> 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.get_toc()*, and later close or change the document, then this cannot and does not change variable *toc* in any way. It is your responsibility to refresh such variables as required.

14.6 Design of Method `Page.show_pdf_page()`

14.6.1 Purpose and Capabilities

The method displays an image of a (“source”) page of another PDF document within a specified rectangle of the current (“containing”, “target”) page.

- **In contrast** to `Page.insert_image()`, this display is vector-based and hence remains accurate across zooming levels.
- **Just like** `Page.insert_image()`, the size of the display is adjusted to the given rectangle.

The following variations of the display are currently supported:

- Bool parameter `keep_proportion` controls whether to maintain the aspect ratio (default) or not.
- Rectangle parameter `clip` restricts the visible part of the source page rectangle. Default is the full page.
- float `rotation` rotates the display by an arbitrary angle (degrees). If the angle is not an integer multiple of 90, only 2 of the 4 corners may be positioned on the target border if also `keep_proportion` is true.
- Bool parameter `overlay` controls whether to put the image on top (foreground, default) of current page content or not (background).

Use cases include (but are not limited to) the following:

1. “Stamp” a series of pages of the current document with the same image, like a company logo or a watermark.
2. Combine arbitrary input pages into one output page to support “booklet” or double-sided printing (known as “4-up”, “n-up”).
3. Split up (large) input pages into several arbitrary pieces. This is also called “posterization”, because you e.g. can split an A4 page horizontally and vertically, print the 4 pieces enlarged to separate A4 pages, and end up with an A2 version of your original page.

14.6.2 Technical Implementation

This is done using PDF “**Form XObjects**”, see section 8.10 on page 217 of *Adobe PDF References*. On execution of a `Page.show_pdf_page(rect, src, pno, ...)`, the following things happen:

1. The `resources` and `contents` objects of page `pno` in document `src` are copied over to the current document, jointly creating a new **Form XObject** with the following properties. The PDF `xref` number of this object is returned by the method.
 - a. `/BBox` equals `/Mediabox` of the source page
 - b. `/Matrix` equals the identity matrix `[1 0 0 1 0 0]`
 - c. `/Resources` equals that of the source page. This involves a “deep-copy” of hierarchically nested other objects (including fonts, images, etc.). The complexity involved here is covered by MuPDF’s grafting¹ technique functions.

¹ MuPDF supports “deep-copying” objects between PDF documents. To avoid duplicate data in the target, it uses so-called “graftmaps”, like a form of scratchpad: for each object to be copied, its `xref` number is looked up in the graftmap. If found, copying is skipped. Otherwise, the new `xref` is recorded and the copy takes place. PyMuPDF makes use of this technique in two places so far: `Document.insert_pdf()` and `Page.show_pdf_page()`. This process is fast and very efficient, because it prevents multiple copies of typically large and frequently referenced data, like images and fonts. However, you may still want to consider using garbage collection (option 4) in any of the following cases:

1. The target PDF is not new / empty: grafting does not check for resources that already existed (e.g. images, fonts) in the target document before

- d. This is a stream object type, and its stream is an exact copy of the combined data of the source page's */Contents* objects.

This step is only executed once per shown source page. Subsequent displays of the same page only create pointers (done in next step) to this object.

2. A second **Form XObject** is then created which the target page uses to invoke the display. This object has the following properties:
 - a. */BBox* equals the */CropBox* of the source page (or *clip*).
 - b. */Matrix* represents the mapping of */BBox* to the target rectangle.
 - c. */XObject* references the previous XObject via the fixed name *fullpage*.
 - d. The stream of this object contains exactly one fixed statement: */fullpage Do*.
3. The *resources* and *contents* objects of the target page are now modified as follows.
 - a. Add an entry to the */XObject* dictionary of */Resources* with the name *fzFrm<n>* (with *n* chosen such that this entry is unique on the page).
 - b. Depending on *overlay*, prepend or append a new object to the page's */Contents* array, containing the statement *q /fzFrm<n> Do Q*.

14.7 Redirecting Error and Warning Messages

Since MuPDF version 1.16 error and warning messages can be redirected via an official plugin.

PyMuPDF will put error messages to *sys.stderr* prefixed with the string "mupdf:". Warnings are internally stored and can be accessed via *fitz.TOOLS.mupdf_warnings()*. There also is a function to empty this store.

opening it.

2. Using *Page.show_pdf_page()* for more than one source document: each grafting occurs **within one source** PDF only, not across multiple. So if e.g. the same image exists in pages from different source PDFs, then this will not be detected until garbage collection.

CHANGE LOG

Changes in Version 1.21.1 (2022-12-13)

- This release uses MuPDF-1.21.1.
- Bug fixes:
 - **Fixed #2110:** Fully embedded font is extracted only partially if it occupies more than one object
 - **Fixed #2094:** Rectangle Detection Logic
 - **Fixed #2088:** Destination point not set for named links in toc
 - **Fixed #2087:** Image with Filter “[/FlateDecode/JPXDecode]” not extracted
 - **Fixed #2086:** Document.save() owner_pw & user_pw has buffer overflow bug
 - **Fixed #2076:** Segfault in fitz.py
 - **Fixed #2057:** Document.save garbage parameter not working in PyMuPDF 1.21.0
 - **Fixed #2051:** Missing DPI Parameter
 - **Fixed #2048:** Invalid size of TextPage and bbox with newest version 1.21.0
 - **Fixed #2045:** SystemError: <built-in function Page_get_texttrace> returned a result with an error set
 - **Fixed #2039:** 1.21.0 fails to build against system libmupdf
 - **Fixed #2036:** Archive::Archive defined twice
- Other
 - Swallow “&zoom=nan” in link uri strings.
 - Add new Page utility methods `Page.replace_image()` and `Page.delete_image()`.
- Documentation:
 - **#2040:** Added note about test failure with non-default build of MuPDF, to `tests/README.md`.
 - **#2037:** In `docs/installation.rst`, mention incompatibility with chocolatey.org on Windows.
 - **#2061:** Fixed description of `Annot.file_info`.
 - **#2065:** Show how to insert internal PDF link.
 - Improved description of building from source without an sdist.
 - Added information about running tests.
 - **#2084:** Fixed broken link to PyMuPDF-Utilities.

Changes in Version 1.21.0 (2022-11-8)

- This release uses MuPDF-1.21.0.
- New feature: Stories.
- Added wheels for Python-3.11.
- Bug fixes:
 - **Fixed #1701:** Broken custom image insertion.
 - **Fixed #1854:** *Document.delete_pages()* declines keyword arguments.
 - **Fixed #1868:** Access Violation Error at *page.apply_redactions()*.
 - **Fixed #1909:** Adding text with *fontname="Helvetica"* can silently fail.
 - **Fixed #1913:** *draw_rect()*: does not respect width if color is not specified.
 - **Fixed #1917:** *subset_fonts()*: make it possible to silence the stdout.
 - **Fixed #1936:** Rectangle detection can be incorrect producing wrong output.
 - **Fixed #1945:** Segmentation fault when saving with *clean=True*.
 - **Fixed #1965:** *pdfocr_save()* Hard Crash.
 - **Fixed #1971:** Segmentation fault when using *get_drawings()*.
 - **Fixed #1946:** *block_no* and *block_type* switched in *get_text()* docs.
 - **Fixed #2013:** AttributeError: 'Widget' object has no attribute '_annot' in delete widget.
- Misc changes to core code:
 - Fixed various compiler warnings and a sequence-point bug.
 - Added support for Memento builds.
 - Fixed leaks detected by Memento in test suite.
 - Fixed handling of exceptions in *set_name()* and *set_rect()*.
 - Allow build with latest MuPDF, for regular testing of PyMuPDF master.
 - Cope with new MuPDF exceptions when setting rect for some Annot types.
 - Reduced cosmetic differences between MuPDF's config.h and PyMuPDF's _config.h.
 - Cope with various changes to MuPDF API.
- Other:
 - Fixed various broken links and typos in docs.
 - Mention install of *swig-python* on MacOS for #875.
 - Added (untested) wheels for macos-arm64.

Changes in Version 1.20.2

- This release uses MuPDF-1.20.3.
- **Fixed #1787.** Fix linking issues on Unix systems.
- **Fixed #1824.** SegFault when applying redactions overlapping a transparent image. (Fixed in MuPDF-1.20.3.)
- Improvements to documentation:
 - Improved information about building from source in docs/installation.rst.
 - Clarified memory allocation setting JM_MEMORY` in ``docs/tools.rst.

- Fixed link to PDF Reference manual in docs/app3.rst.
- Fixed building of html documentation on OpenBSD.
- Moved old docs/faq.rst into separate docs/recipes-* files.
- Removed some unused files and directories:
 - installation/
 - docs/wheelnames.txt

Changes in Version 1.20.1

- **Fixed #1724.** Fix for building on FreeBSD.
- **Fixed #1771.** *linkDest()* had a broken call to *re.match()*, introduced in 1.20.0.
- **Fixed #1751.** *get_drawings()* and *get_cdrawings()* previously always returned with *closePath=False*.
- **Fixed #1645.** Default FreeText annotation text color is now black.
- Improvements to sphinx-generated documentation:
 - Use readthedocs theme with enhancements.
 - Renamed the .txt files to have .rst suffixes.

Changes in Version 1.20.0

This release uses MuPDF-1.20.0, released 2022-06-15.

- Cope with new MuPDF link uri format, changed from *#<int>,<int>,<int>* to *#page=<int>&zoom=<float>,<float>,<float>*.
- In tests/test_insertpdf.py, use new reference output joined-1.20.pdf. We also check that new output values are approximately the same as the old ones.
- **Fixed #1738.** Leak of *pdf_graft_map*. Also fixed a SEGV issue that this seemed to expose, caused by incorrect freeing of underlying fz_document.
- **Fixed #1733.** Fixed ownership of *Annotation.get_pixmap()*.

Changes to build/release process:

- If pip builds from source because an appropriate wheel is not available, we no longer require MuPDF to be pre-installed. Instead the required MuPDF source is embedded in the sdist and automatically built into PyMuPDF.
- Various changes to setup.py to download the required MuPDF release as required. See comments at start of setup.py for details.
- Added .github/workflows/build_wheels.yml to control building of wheels on Github.

Changes in Version 1.19.6

- **Fixed #1620.** The *TextPage* created by *Page.get_textpage()* will now be freed correctly (removed memory leak).
- **Fixed #1601.** Document open errors should now be more concise and easier to interpret. In the course of this, two PyMuPDF-specific Python exceptions have been **added**:
 - *EmptyFileError* – raised when trying to create a *Document* (*fitz.open()*) from an empty file or zero-length memory.
 - *FileDataError* – raised when MuPDF encounters irrecoverable document structure issues.

- **Added** `Page.load_widget()` given a PDF field's xref.
 - **Added** Dictionary `pdfcolor` which provide the about 500 colors defined as PDF color values with the lower case color name as key.
 - **Added** algebra functionality to the `Quad` class. These objects can now also be added and subtracted among themselves, and be multiplied by numbers and matrices.
 - **Added** new constants defining the default text extraction flags for more comfortable handling. Their naming convention is like `TEXTFLAGS_WORDS` for `page.get_text("words")`. See *Text Extraction Flags Defaults*.
 - **Changed** `Page.annots()` and `Page.widgets()` to detect and prevent reloading the page (illegally) inside the iterator loops via `Document.reload_page()`. Doing this brings down the interpreter. Documented clean ways to do annotation and widget mass updates within properly designed loops.
 - **Changed** several internal utility functions to become standalone ("SWIG inline") as opposed to be part of the `Tools` class. This, among other things, increases the performance of geometry object creation.
 - **Changed** `Document.update_stream()` to always accept stream updates - whether or not the dictionary object behind the xref already is a stream. Thus the former `new` parameter is now ignored and will be removed in v1.20.0.
-

Changes in Version 1.19.5

- **Fixed** #1518. A limited "fix": in some cases, rectangles and quadrupels were not correctly encoded to support re-drawing by *Shape*.
 - **Fixed** #1521. This had the same ultimate reason behind issue #1510.
 - **Fixed** #1513. Some Optional Content functions did not support non-ASCII characters.
 - **Fixed** #1510. Support more soft-mask image subtypes.
 - **Fixed** #1507. Immunize against items in the outlines chain, that are "null" objects.
 - **Fixed** re-opened #1417. ("too many open files"). This was due to insufficient calls to MuPDF's `fz_drop_document()`. This also fixes #1550.
 - **Fixed** several undocumented issues in relation to incorrectly setting the text span origin *point_like*.
 - **Fixed** undocumented error computing the character bbox in method `Page.get_texttrace()` when text is **flipped** (as opposed to just rotated).
 - **Added** items to the dictionary returned by `image_properties()`: `orientation` and `transform` report the natural image orientation (EXIF data).
 - **Added** method `Document.xref_copy()`. It will make a given target PDF object an exact copy of a source object.
-

Changes in Version 1.19.4

- **Fixed** #1505. Immunize against circular outline items.
- **Fixed** #1484. Correct CropBox coordinates are now returned in all situations.
- **Fixed** #1479.
- **Fixed** #1474. TextPage objects are now properly deleted again.
- **Added** `Page` methods and attributes for PDF `/ArtBox`, `/BleedBox`, `/TrimBox`.
- **Added** global attribute `TESSDATA_PREFIX` for easy checking of OCR support.

- **Changed** `Document.xref_set_key()` such that dictionary keys will physically be removed if set to value "null".
 - **Changed** `Document.extract_font()` to optionally return a dictionary (instead of a tuple).
-

Changes in Version 1.19.3

This patch version implements minor improvements for `Pixmap` and also some important fixes.

- **Fixed** #1351. Reverted code that introduced the memory growth in v1.18.15.
 - **Fixed** #1417. Developed circumvention for growth of open file handles using `Document.insert_pdf()`.
 - **Fixed** #1418. Developed circumvention for memory growth using `Document.insert_pdf()`.
 - **Fixed** #1430. Developed circumvention for mass pixmap generations of document pages.
 - **Fixed** #1433. Solves a bbox error for some Type 3 font in PyMuPDF text processing.
 - **Added** `Pixmap.color_topusage()` to determine the share of the most frequently used color. Solves #1397.
 - **Added** `Pixmap.warp()` which makes a new pixmap from a given arbitrary convex quad inside the pixmap.
 - **Added** `Annot.irt_xref` and `Annot.set_irt_xref()` to inquire or set the `/IRT` ("In Responde To") property of an annotation. Implements #1450.
 - **Added** `Rect.torect()` and `IRect.torect()` which compute a matrix that transforms to a given other rectangle.
 - **Changed** `Pixmap.color_count()` to also return the count of each color.
 - **Changed** `Page.get_texttrace()` to also return correct span and character bboxes if `span["dir"] != (1, 0)`.
-

Changes in Version 1.19.2

This patch version implements minor improvements for `Page.get_drawings()` and also some important fixes.

- **Fixed** #1388. Fixed intermittent memory corruption when insert or updating annotations.
 - **Fixed** #1375. Inconsistencies between line numbers as returned by the "words" and the "dict" options of `Page.get_text()` have been corrected.
 - **Fixed** #1364. The check for being a "rawdict" span in `recover_span_quad()` now works correctly.
 - **Fixed** #1342. Corrected the check for rectangle infiniteness in `Page.show_pdf_page()`.
 - **Changed** `Page.get_drawings()`, `Page.get_cdrawings()` to return an indicator on the area orientation covered by a rectangle. This implements #1355. Also, the recognition rate for rectangles and quads has been significantly improved.
 - **Changed** all text search and extraction methods to set the new `flags` option `TEXT_MEDIABOX_CLIP` to ON by default. That bit causes the automatic suppression of all characters that are completely outside a page's mediabox (in as far as that notion is supported for a document type). This eliminates the need for using `clip=page.rect` or similar for omitting text outside the visible area.
 - **Added** parameter "dpi" to `Page.get_pixmap()` and `Annot.get_pixmap()`. When given, parameter "matrix" is ignored, and a `Pixmap` with the desired dots per inch is created.
 - **Added** attributes `Pixmap.is_monochrome` and `Pixmap.is_unicolor` allowing fast checks of pixmap properties. Addresses #1397.
 - **Added** method `Pixmap.color_count()` to determine the unique colors in the pixmap.
-

- **Added** boolean parameter "compress" to PDF document method `Document.update_stream()`. Addresses / enables solution for #1408.

Changes in Version 1.19.1

This is the first patch version to support MuPDF v1.19.0. Apart from one bug fix, it includes important improvements for OCR support and the option to **sort extracted text** to the standard reading order "from top-left to bottom-right".

- **Fixed** #1328. "words" text extraction again returns correct (x0, y0) coordinates.
- **Changed** `Page.get_textpage_ocr()`: it now supports parameter `dpi` to control OCR quality. It is also possible to choose whether the **full page** should be OCR'd or **only the images displayed** by the page.
- **Changed** `Page.get_drawings()` and `Page.get_cdrawings()` to automatically convert colors to RGB color tuples. Implements #1332. Similar change was applied to `Page.get_texttrace()`.
- **Changed** `Page.get_text()` to support a parameter `sort`. If set to `True` the output is conveniently sorted.

Changes in Version 1.19.0

This is the first version supporting MuPDF 1.19.*, published 2021-10-05. It introduces many new features compared to the previous version 1.18.*.

PyMuPDF has now picked up integrated Tesseract OCR support, which was already present in MuPDF v1.18.0.

- Supported images can be OCR'd via their *Pixmap* which results in a 1-page PDF with a text layer.
- All supported document pages (i.e. not only PDFs), can be OCR'd using specialized text extraction methods. The result is a mixture of standard and OCR text (depending on which part of the page was deemed to require OCR'ing) that can be searched and extracted without restrictions.
- All this requires an independent installation of Tesseract. MuPDF actually (only) needs the location of Tesseract's "tessdata" folder, where its language support data are stored. This location must be available as environment variable `TESSDATA_PREFIX`.

A new MuPDF feature is **journalling PDF updates**, which is also supported by this PyMuPDF version. Changes may be logged, rolled back or replayed, allowing to implement a whole new level of control over PDF document integrity – similar to functions present in modern database systems.

A third feature (unrelated to the new MuPDF version) includes the ability to detect when page **objects cover or hide each other**. It is now e.g. possible to see that text is covered by a drawing or an image.

- **Changed** terminology and meaning of important geometry concepts: Rectangles are now characterized as *finite*, *valid* or *empty*, while the definitions of these terms have also changed. Rectangles specifically are now thought of being "open": not all corners and sides are considered part of the rectangle. Please do read the *Rect* section for details.
- **Added** new parameter "`no_new_id`" to `Document.save()` / `Document.tobytes()` methods. Use it to suppress updating the second item of the document /ID which in PDF indicates that the original file has been updated. If the PDF has no /ID at all yet, then no new one will be created either.
- **Added** a **journalling facility** for PDF updates. This allows logging changes, undoing or redoing them, or saving the journal for later use. Refer to `Document.journal_enable()` and friends.
- **Added** new *Pixmap* methods `Pixmap.pdfocr_save()` and `Pixmap.pdfocr_tobytes()`, which generate a 1-page PDF containing the pixmap as PNG image with OCR text layer.
- **Added** `Page.get_textpage_ocr()` which executes optical character recognition for the page, then extracts the results and stores them together with "normal" page content in a *TextPage*. Use or reuse this object in subsequent text extractions and text searches to avoid multiple efforts. The existing text search and text extraction methods have been extended to support a separately created textpage – see next item.

- **Added** a new parameter `textpage` to text extraction and text search methods. This allows reuse of a previously created `TextPage` and thus achieves significant runtime benefits – which is especially important for the new OCR features. But “normal” text extractions can definitely also benefit.
 - **Added** `Page.get_texttrace()`, a technical method delivering low-level text character properties. It was present before as a private method, but the author felt it now is mature enough to be officially available. It specifically includes a “sequence number” which indicates the page appearance build operation that painted the text.
 - **Added** `Page.get_bboxlog()` which delivers the list of rectangles of page objects like text, images or drawings. Its significance lies in its sequence: rectangles intersecting areas with a lower index are covering or hiding them.
 - **Changed** methods `Page.get_drawings()` and `Page.get_cdrawings()` to include a “sequence number” indicating the page appearance build operation that created the drawing.
 - **Fixed** #1311. Field values in comboboxes should now be handled correctly.
 - **Fixed** #1290. Error was caused by incorrect rectangle emptiness check, which is fixed due to new geometry logic of this version.
 - **Fixed** #1286. Text alignment for redact annotations is working again.
 - **Fixed** #1287. Infinite loop issue for non-Windows systems when applying some redactions has been resolved.
 - **Fixed** #1284. Text layout destruction after applying redactions in some cases has been resolved.
-

Changes in Version 1.18.18 / 1.18.19

- **Fixed** issue #1266. Failure to set `Pixmap.samples` in important cases, was hotfixed in a new version 1.18.19.
 - **Fixed** issue #1257. Removing the read-only flag from PDF fields is now possible.
 - **Fixed** issue #1252. Now correctly specifying the zoom value for PDF link annotations.
 - **Fixed** issue #1244. Now correctly computing the transform matrix in `Page.get_image__bbox()`.
 - **Fixed** issue #1241. Prevent returning artifact characters in `Page.get_textbox()`, which happened in certain constellations.
 - **Fixed** issue #1234. Avoid creating infinite rectangles in corner cases – `Page.get_drawings()`, `Page.get_cdrawings()`.
 - **Added** test data and test scripts to the source PyPI source distribution.
-

Changes in Version 1.18.17

Focus of this version are major performance improvements of selected functions.

- **Fixed** issue #1199. Using a non-existing page number in `Document.get_page_images()` and friends will no longer lead to segfaults.
 - **Changed** `Page.get_drawings()` to now differentiate between “stroke”, “fill” and combined paths. Paths containing more than one rectangle (i.e. “re” items) are now supported. Extracting “clipped” paths is now available as an option.
 - **Added** `Page.get_cdrawings()`, performance-optimized version of `Page.get_drawings()`.
 - **Added** `Pixmap.samples_mv`, *memoryview* of a pixmap’s pixel area. Does not copy and thus always accesses the current state of that area.
 - **Added** `Pixmap.samples_ptr`, Python “pointer” to a pixmap’s pixel area. Allows much faster creation (factor 800+) of Qt images.
-

Changes in Version 1.18.16

- **Fixed** issue #1184. Existing PDF widget fonts in a PDF are now accepted (i.e. not forcedly changed to a Base-14 font).
 - **Fixed** issue #1154. Text search hits should now be correct when clip is specified.
 - **Fixed** issue #1152.
 - **Fixed** issue #1146.
 - **Added** `Link.flags` and `Link.set_flags()` to the `Link` class. Implements enhancement requests #1187.
 - **Added** option to `simulate TextWriter.fill_textbox()` output for predicting the number of lines, that a given text would occupy in the textbox.
 - **Added** text output support as subcommand `gettext` to the `fitz` CLI module. Most importantly, original **physical text layout** reproduction is now supported.
-

Changes in Version 1.18.15

- **Fixed** issue #1088. Removing an annotation's fill color should now work again both ways, using the `fill_color=[]` argument in `Annot.update()` as well as `fill=[]` in `Annot.set_colors()`.
 - **Fixed** issue #1081. `Document.subset_fonts()`: fixed an error which created wrong character widths for some fonts.
 - **Fixed** issue #1078. `Page.get_text()` and other methods related to text extraction: changed the default value of the `TextPage` `flags` parameter. All whitespace and ligatures are now preserved.
 - **Fixed** issue #1085. The old `snake_cased` alias of `fitz.detTextlength` is now defined correctly.
 - **Changed** `Document.subset_fonts()` will now correctly prefix font subsets with an appropriate six letter uppercase tag, complying with the PDF specification.
 - **Added** new method `Widget.button_states()` which returns the possible values that a button-type field can have when being set to “on” or “off”.
 - **Added** support of text with **Small Capital** letters to the `Font` and `TextWriter` classes. This is reflected by an additional bool parameter `small_caps` in various of their methods.
-

Changes in Version 1.18.14

- **Finished** implementing new, “snake_cased” names for methods and properties, that were “camelCased” and awkward in many aspects. At the end of this documentation, there is section *Deprecated Names* with more background and a mapping of old to new names.
- **Fixed** issue #1053. `Page.insert_image()`: when given, include image mask in the hash computation.
- **Fixed** issue #1043. Added `Pixmap.getPNGdata` to the aliases of `Pixmap.tobytes()`.
- **Fixed** an internal error when computing the envelopping rectangle of drawn paths as returned by `Page.get_drawings()`.
- **Fixed** an internal error occasionally causing loops when outputting text via `TextWriter.fill_textbox()`.
- **Added** `Font.char_lengths()`, which returns a tuple of character widths of a string.
- **Added** more ways to specify pages in `Document.delete_pages()`. Now a sequence (list, tuple or range) can be specified, and the Python `del` statement can be used. In the latter case, Python slices are also accepted.

- **Changed** `Document.del_toc_item()`, which disables a single item of the TOC: previously, the title text was removed. Instead, now the complete item will be shown grayed-out by supporting viewers.
-

Changes in Version 1.18.13

- **Fixed** issue #1014.
 - **Fixed** an internal memory leak when computing image bboxes – `Page.get_image_bbox()`.
 - **Added** support for low-level access and modification of the PDF trailer. Applies to `Document.xref_get_keys()`, `Document.xref_get_key()`, and `Document.xref_set_key()`.
 - **Added** documentation for maintaining private entries in PDF metadata.
 - **Added** documentation for handling transparent image insertions, `Page.insert_image()`.
 - **Added** `Page.get_image_rects()`, an improved version of `Page.get_image_bbox()`.
 - **Changed** `Document.delete_pages()` to support various ways of specifying pages to delete. Implements #1042.
 - **Changed** `Page.insert_image()` to also accept the xref of an existing image in the file. This allows “copying” images between pages, and extremely fast multiple insertions.
 - **Changed** `Page.insert_image()` to also accept the integer parameter `alpha`. To be used for performance improvements.
 - **Changed** `Pixmap.set_alpha()` to support new parameters for pre-multiplying colors with their alpha values and setting a specific color to fully transparent (e.g. white).
 - **Changed** `Document.embfile_add()` to automatically set creation and modification date-time. Correspondingly, `Document.embfile_upd()` automatically maintains modification date-time (/ModDate PDF key), and `Document.embfile_info()` correspondingly reports these data. In addition, the embedded file’s associated “collection item” is included via its `xref`. This supports the development of PDF portfolio applications.
-

Changes in Version 1.18.11 / 1.18.12

- **Fixed** issue #972. Improved layout of source distribution material.
 - **Fixed** issue #962. Stabilized Linux distribution detection for generating PyMuPDF from sources.
 - **Added:** `Page.get_xobjects()` delivers the result of `Document.get_page_xobjects()`.
 - **Added:** `Page.get_image_info()` delivers meta information for all images shown on the page.
 - **Added:** `Tools.mupdf_display_warnings()` allows setting on / off the display of MuPDF-generated warnings. The default is off.
 - **Added:** `Document.ez_save()` convenience alias of `Document.save()` with some different defaults.
 - **Changed:** Image extractions of document pages now also contain the image’s **transformation matrix**. This concerns `Page.get_image_bbox()` and the DICT, JSON, RAWDICT, and RAWJSON variants of `Page.get_text()`.
-

Changes in Version 1.18.10

- **Fixed** issue #941. Added old aliases for `DisplayList.get_pixmap()` and `DisplayList.get_textpage()`.
 - **Fixed** issue #929. Stabilized removal of JavaScript objects with `Document.scrub()`.
 - **Fixed** issue #927. Removed a loop in the reworked `TextWriter.fill_textbox()`.
-

- **Changed** `Document.xref_get_keys()` and `Document.xref_get_key()` to also allow accessing the PDF trailer dictionary. This can be done by using `-1` as the xref number argument.
 - **Added** a number of functions for reconstructing the quads for text lines, spans and characters extracted by `Page.get_text()` options “dict” and “rawdict”. See `recover_quad()` and friends.
 - **Added** `Tools.unset_quad_corrections()` to suppress character quad corrections (occasionally required for erroneous fonts).
-

Changes in Version 1.18.9

- **Fixed** issue #888. Removed ambiguous statements concerning PyMuPDF’s license, which is now clearly stated to be GNU AGPL V3.
 - **Fixed** issue #895.
 - **Fixed** issue #896. Since v1.17.6 PyMuPDF suppresses the font subset tags and only reports the base fontname in text extraction outputs “dict” / “json” / “rawdict” / “rawjson”. Now a new global parameter can request the old behaviour, `Tools.set_subset_fontnames()`.
 - **Fixed** issue #885. Pixmap creation now also works with filenames given as `pathlib.Paths`.
 - **Changed** `Document.subset_fonts()`: Text is **not rewritten** any more and should therefore **retain all its original properties** – like being hidden or being controlled by Optional Content mechanisms.
 - **Changed** `TextWriter` output to also accept text in right to left mode (Arabic, Hebrew): `TextWriter.fill_textbox()`, `TextWriter.append()`. These methods now accept a new boolean parameter `right_to_left`, which is `False` by default. Implements #897.
 - **Changed** `TextWriter.fill_textbox()` to return all lines of text, that did not fit in the given rectangle. Also changed the default of the `warn` parameter to no longer print a warning message in overflow situations.
 - **Added** a utility function `recover_quad()`, which computes the quadrilateral of a span. This function can be used for correctly marking text extracted with the “dict” or “rawdict” options of `Page.get_text()`.
-

Changes in Version 1.18.8

This is a bug fix version only. We are publishing early because of the potentially widely used functions.

- **Fixed** issue #881. Fixed a memory leak in `Page.insert_image()` when inserting images from files or memory.
 - **Fixed** issue #878. `pathlib.Path` objects should now correctly handle file path hierarchies.
-

Changes in Version 1.18.7

- **Added** an experimental `Document.subset_fonts()` which reduces the size of eligible fonts based on their use by text in the PDF. Implements #855.
- **Implemented** request #870: `Document.convert_to_pdf()` now also supports PDF documents.
- **Renamed** `Document.write` to `Document.tobytes()` for greater clarity. But the deprecated name remains available for some time.
- **Implemented** request #843: `Document.tobytes()` now supports linearized PDF output. `Document.save()` now also supports writing to Python **file objects**. In addition, the `open` function now also supports Python file objects.
- **Fixed** issue #844.
- **Fixed** issue #838.

- **Fixed** issue #823. More logic for better support of OCR'd text output (Tesseract, ABBYY).
- **Fixed** issue #818.
- **Fixed** issue #814.
- **Added** `Document.get_page_labels()` which returns a list of page label definitions of a PDF.
- **Added** `Document.has_annots()` and `Document.has_links()` to check whether these object types are present anywhere in a PDF.
- **Added** expert low-level functions to simplify inquiry and modification of PDF object sources: `Document.xref_get_keys()` lists the keys of object `xref`, `Document.xref_get_key()` returns type and content of a key, and `Document.xref_set_key()` modifies the key's value.
- **Added** parameter `thumbnails` to `Document.scrub()` to also allow removing page thumbnail images.
- **Improved** documentation for how to add valid text marker annotations for non-horizontal text.

We continued the process of renaming methods and properties from “*mixedCase*” to “*snake_case*”. Documentation usually mentions the new names only, but old, deprecated names remain available for some time.

Changes in Version 1.18.6

- **Fixed** issue #812.
- **Fixed** issue #793. Invalid document metadata previously prevented opening some documents at all. This error has been removed.
- **Fixed** issue #792. Text search and text extraction will make no rectangle containment checks at all if the default `clip=None` is used.
- **Fixed** issue #785.
- **Fixed** issue #780. Corrected a parameter check error.
- **Fixed** issue #779. Fixed typo
- **Added** an option to set the desired line height for text boxes. Implements #804.
- **Changed** text position retrieval to better cope with Tesseract's glyphless font. Implements #803.
- **Added** an option to choose the prefix of new annotations, fields and links for providing unique annotation ids. Implements request #807.
- **Added** getting and setting color and text properties for Table of Contents items for PDFs. Implements #779.
- **Added** PDF page label handling: `Page.get_label()` returns the page label, `Document.get_page_numbers()` return all page numbers having a specified label, and `Document.set_page_labels()` adds or updates a PDF's page label definition.

Note: This version introduces **Python type hinting**. The goal is to provide each parameter and the return value of all functions and methods with type information. This still is work in progress although the majority of functions has already been handled.

Changes in Version 1.18.5

Apart from several fixes, this version also focusses on several minor, but important feature improvements. Among the latter is a more precise computation of proper line heights and insertion points for writing / inserting text. As opposed to using font-agnostic constants, these values are now taken from the font's properties.

Also note that this is the first version which does no longer provide pregenerated wheels for Python versions older than 3.6. PIP also discontinues support for these by end of this year 2020.

- **Fixed** issue #771. By using “small glyph heights” option, the full page text can be extracted.
 - **Fixed** issue #768.
 - **Fixed** issue #750.
 - **Fixed** issue #739. The “dict”, “rawdict” and corresponding JSON output variants now have two new *span* keys: “ascender” and “descender”. These floats represent special font properties which can be used to compute bboxes of spans or characters of **exactly fontsize height** (as opposed to the default line height). An example algorithm is shown in section “Span Dictionary” [here](#). Also improved the detection and correction of ill-specified ascender / descender values encountered in some fonts.
 - **Added** a new, experimental `Tools.set_small_glyph_heights()` – also in response to issue #739. This method sets or unsets a global parameter to **always compute bboxes with fontsize height**. If “on”, text searching and all text extractions will returned rectangles, bboxes and quads with a smaller height.
 - **Fixed** issue #728.
 - **Changed** fill color logic of ‘Polyline’ annotations: this parameter now only pertains to line end symbols – the annotation itself can no longer have a fill color. Also addresses issue #727.
 - **Changed** `Page.getImageBbox()` to also compute the bbox if the image is contained in an XObject.
 - **Changed** `Shape.insertTextbox()`, resp. `Page.insertTextbox()`, resp. `TextWriter.fillTextbox()` to respect font’s properties “ascender” / “descender” when computing line height and insertion point. This should no longer lead to line overlaps for multi-line output. These methods used to ignore font specifics and used constant values instead.
-

Changes in Version 1.18.4

This version adds several features to support PDF Optional Content. Among other things, this includes OCMDs (Optional Content Membership Dictionaries) with the full scope of “*visibility expressions*” (PDF key /VE), text insertions (including the `TextWriter` class) and drawings.

- **Fixed** issue #727. Freetext annotations now support an uncolored rectangle when `fill_color=None`.
- **Fixed** issue #726. UTF-8 encoding errors are now handled for HTML / XML `Page.getText()` output.
- **Fixed** issue #724. Empty values are no longer stored in the PDF /Info metadata dictionary.
- **Added** new methods `Document.set_oc()` and `Document.get_oc()` to set or get optional content references for **existing** image and form XObjects. These methods are similar to the same-named methods of `Annot`.
- **Added** `Document.set_ocmd()`, `Document.get_ocmd()` for handling OCMDs.
- **Added Optional Content** support for text insertion and drawing.
- **Added** new method `Page.deleteWidget()`, which deletes a form field from a page. This is analogous to deleting annotations.
- **Added** support for Popup annotations. This includes defining the Popup rectangle and setting the Popup to open or closed. Methods / attributes `Annot.set_popup()`, `Annot.set_open()`, `Annot.has_popup`, `Annot.is_open`, `Annot.popup_rect`, `Annot.popup_xref`.

Other changes:

- The **naming of methods and attributes** in PyMuPDF is far from being satisfactory: we have *CamelCases*, *mixedCases* and *lower_case_with_underscores* all over the place. With the `Annot` as the first candidate, we have started an activity to clean this up step by step, converting to lower case with underscores for methods and attributes while keeping UPPERCASE for the constants.

- Old names will remain available to prevent code breaks, but they will no longer be mentioned in the documentation.
 - New methods and attributes of all classes will be named according to the new standard.
-

Changes in Version 1.18.3

As a major new feature, this version introduces support for PDF's **Optional Content** concept.

- **Fixed** issue #714.
 - **Fixed** issue #711.
 - **Fixed** issue #707: if a PDF user password, but no owner password is supplied nor present, then the user password is also used as the owner password.
 - **Fixed** `expand` and `deflate` parameters of methods `Document.save()` and `Document.write()`. Individual image and font compression should now finally work. Addresses issue #713.
 - **Added** a support of PDF optional content. This includes several new `Document` methods for inquiring and setting optional content status and adding optional content configurations and groups. In addition, images, form XObjects and annotations now can be bound to optional content specifications. **Resolved** issue #709.
-

Changes in Version 1.18.2

This version contains some interesting improvements for text searching: any number of search hits is now returned and the `hit_max` parameter was removed. The new `clip` parameter in addition allows to restrict the search area. Searching now detects hyphenations at line breaks and accordingly finds hyphenated words.

- **Fixed** issue #575: if using `quads=False` in text searching, then overlapping rectangles on the same line are joined. Previously, parts of the search string, which belonged to different “marked content” items, each generated their own rectangle – just as if occurring on separate lines.
 - **Added** `Document.isRepaired`, which is true if the PDF was repaired on open.
 - **Added** `Document.setXmlMetadata()` which either updates or creates PDF XML metadata. Implements issue #691.
 - **Added** `Document.getXmlMetadata()` returns PDF XML metadata.
 - **Changed** creation of PDF documents: they will now always carry a PDF identification (/ID field) in the document trailer. Implements issue #691.
 - **Changed** `Page.searchFor()`: a new parameter `clip` is accepted to restrict the search to this rectangle. Correspondingly, the attribute `TextPage.rect` is now respected by `TextPage.search()`.
 - **Changed** parameter `hit_max` in `Page.searchFor()` and `TextPage.search()` is now obsolete: methods will return all hits.
 - **Changed** character **selection criteria** in `Page.getText()`: a character is now considered to be part of a `clip` if its `bbox` is fully contained. Before this, a non-empty intersection was sufficient.
 - **Changed** `Document.scrub()` to support a new option `redact_images`. This addresses issue #697.
-

Changes in Version 1.18.1

- **Fixed** issue #692. PyMuPDF now detects and recovers from more cyclic resource dependencies in PDF pages and for the first time reports them in the MuPDF warnings store.
 - **Fixed** issue #686.
-

- **Added** opacity options for the *Shape* class: Stroke and fill colors can now be set to some transparency value. This means that all *Page* draw methods, methods `Page.insertText()`, `Page.insertTextbox()`, `Shape.finish()`, `Shape.insertText()`, and `Shape.insertTextbox()` support two new parameters: *stroke_opacity* and *fill_opacity*.
 - **Added** new parameter `mask` to `Page.insertImage()` for optionally providing an external image mask. Resolves issue #685.
 - **Added** `Annot.soundGet()` for extracting the sound of an audio annotation.
-

Changes in Version 1.18.0

This is the first PyMuPDF version supporting MuPDF v1.18. The focus here is on extending PyMuPDF's own functionality – apart from bug fixing. Subsequent PyMuPDF patches may address features new in MuPDF.

- **Fixed** issue #519. This upstream bug occurred occasionally for some pages only and seems to be fixed now: page layout should no longer be ruined in these cases.
 - **Fixed** issue #675.
 - Unsuccessful storage allocations should now always lead to exceptions (circumvention of an upstream bug intermittently crashing the interpreter).
 - *Pixmap* size is now based on `size_t` instead of `int` in C and should be correct even for extremely large pixmaps.
 - **Fixed** issue #668. Specification of dashes for PDF drawing insertion should now correctly reflect the PDF spec.
 - **Fixed** issue #669. A major source of memory leakage in `Page.insert_pdf()` has been removed.
 - **Added** keyword “*images*” to `Page.apply_redactions()` for fine-controlling the handling of images.
 - **Added** `Annot.getText()` and `Annot.getTextbox()`, which offer the same functionality as the *Page* versions.
 - **Added** key “*number*” to the block dictionaries of `Page.getText()` / `Annot.getText()` for options “dict” and “rawdict”.
 - **Added** `glyph_name_to_unicode()` and `unicode_to_glyph_name()`. Both functions do not really connect to a specific font and are now independently available, too. The data are now based on the *Adobe Glyph List*.
 - **Added** convenience functions `adobe_glyph_names()` and `adobe_glyph_uniodes()` which return the respective available data.
 - **Added** `Page.getDrawings()` which returns details of drawing operations on a document page. Works for all document types.
 - Improved performance of `Document.insert_pdf()`. Multiple object copies are now also suppressed across multiple separate insertions from the same source. This saves time, memory and target file size. Previously this mechanism was only active within each single method execution. The feature can also be suppressed with the new method bool parameter `final=1`, which is the default.
 - For PNG images created from pixmaps, the resolution (dpi) is now automatically set from the respective *Pixmap.xres* and *Pixmap.yres* values.
-

Changes in Version 1.17.7

- **Fixed** issue #651. An upstream bug causing interpreter crashes in corner case redaction processings was fixed by backporting MuPDF changes from their development repo.
- **Fixed** issue #645. Pixmap top-left coordinates can be set (again) by their own method, `Pixmap.set_origin()`.
- **Fixed** issue #622. `Page.insertImage()` again accepts a *rect_like* parameter.

- **Added** several new methods to improve and speed-up table of contents (TOC) handling. Among other things, TOC items can now be changed or deleted individually – without always replacing the complete TOC. Furthermore, access to some PDF page attributes is now possible without first **loading** the page. This has a very significant impact on the performance of TOC manipulation.
 - **Added** an option to `Document.insert_pdf()` which allows displaying progress messages. Addresses [#640](#).
 - **Added** `Page.getTextbox()` which extracts text contained in a rectangle. In many cases, this should obsolete writing your own script for this type of thing.
 - **Added** new `clip` parameter to `Page.getText()` to simplify and speed up text extraction of page sub areas.
 - **Added** `TextWriter.appendv()` to add text in **vertical write mode**. Addresses issue [#653](#)
-

Changes in Version 1.17.6

- **Fixed** issue [#605](#)
 - **Fixed** issue [#600](#) – text should now be correctly positioned also for pages with a CropBox smaller than MediaBox.
 - **Added** text span dictionary key `origin` which contains the lower left coordinate of the first character in that span.
 - **Added** attribute `Font.buffer`, a `bytes` copy of the font file.
 - **Added** parameter `sanitize` to `Page.cleanContents()`. Allows switching of sanitization, so only syntax cleaning will be done.
-

Changes in Version 1.17.5

- **Fixed** issue [#561](#) – second go: certain `TextWriter` usages with many alternating fonts did not work correctly.
 - **Fixed** issue [#566](#).
 - **Fixed** issue [#568](#).
 - **Fixed** – opacity is now correctly taken from the `TextWriter` object, if not given in `TextWriter.writeText()`.
 - **Added** a new global attribute `fitz_fontdescriptors`. Contains information about usable fonts from repository `pymupdf-fonts`.
 - **Added** `Font.valid_codepoints()` which returns an array of unicode codepoints for which the font has a glyph.
 - **Added** option `text_as_path` to `Page.getSVGImage()`. This implements [#580](#). Generates much smaller SVG files with parseable text if set to `False`.
-

Changes in Version 1.17.4

- **Fixed** issue [#561](#). Handling of more than 10 `Font` objects on one page should now work correctly.
 - **Fixed** issue [#562](#). Annotation pixmaps are no longer derived from the page pixmap, thus avoiding unintended inclusion of page content.
 - **Fixed** issue [#559](#). This **MuPDF** bug is being temporarily fixed with a pre-version of MuPDF's next release.
 - **Added** utility function `repair_mono_font()` for correcting displayed character spacing for some mono-spaced fonts.
 - **Added** utility method `Document.need_appearances()` for fine-controlling Form PDF behavior. Addresses issue [#563](#).
-

- **Added** utility function `sRGB_to_pdf()` to recover the PDF color triple for a given color integer in sRGB format.
 - **Added** utility function `sRGB_to_rgb()` to recover the (R, G, B) color triple for a given color integer in sRGB format.
 - **Added** utility function `make_table()` which delivers table cells for a given rectangle and desired numbers of columns and rows.
 - **Added** support for optional fonts in repository `pymupdf-fonts`.
-

Changes in Version 1.17.3

- **Fixed** an undocumented issue, which prevented fully cleaning a PDF page when using `Page.cleanContents()`.
 - **Fixed** issue #540. Text extraction for EPUB should again work correctly.
 - **Fixed** issue #548. Documentation now includes `LINK_NAMED`.
 - **Added** new parameter to control start of text in `TextWriter.fillTextbox()`. Implements #549.
 - **Changed** documentation of `Page.add_redact_annot()` to explain the usage of non-builtin fonts.
-

Changes in Version 1.17.2

- **Fixed** issue #533.
 - **Added** options to modify ‘Redact’ annotation appearance. Implements #535.
-

Changes in Version 1.17.1

- **Fixed** issue #520.
- **Fixed** issue #525. Vertices for ‘Ink’ annots should now be correct.
- **Fixed** issue #524. It is now possible to query and set rotation for applicable annotation types.

Also significantly improved inline documentation for better support of interactive help.

Changes in Version 1.17.0

This version is based on MuPDF v1.17. Following are highlights of new and changed features:

- **Added** extended language support for annotations and widgets: a mixture of Latin, Greece, Russian, Chinese, Japanese and Korean characters can now be used in ‘FreeText’ annotations and text widgets. No special arrangement is required to use it.
- Faster page access is implemented for documents supporting a “chapter” structure. This applies to EPUB documents currently. This comes with several new *Document* methods and changes for `Document.loadPage()` and the “indexed” page access `doc[n]`: In addition to specifying a page number as before, a tuple (*chapter*, *pno*) can be specified to identify the desired page.
- **Changed:** Improved support of redaction annotations: images overlapped by redactions are **permanantly modified** by erasing the overlap areas. Also links are removed if overlapped by redactions. This is now fully in sync with PDF specifications.

Other changes:

- **Changed** `TextWriter.writeText()` to support the “*morph*” parameter.
 - **Added** methods `Rect.morph()`, `IRect.morph()`, and `Quad.morph()`, which return a new *Quad*.
-

- **Changed** `Page.add_freetext_annot()` to support text alignment via a new “*align*” parameter.
- **Fixed** issue #508. Improved image rectangle calculation to hopefully deliver correct values in most if not all cases.
- **Fixed** issue #502.
- **Fixed** issue #500. `Document.convertToPDF()` should no longer cause memory leaks.
- **Fixed** issue #496. Annotations and widgets / fields are now added or modified using the coordinates of the **unrotated page**. This behavior is now in sync with other methods modifying PDF pages.
- **Added** `Page.rotationMatrix` and `Page.derotationMatrix` to support coordinate transformations between the rotated and the original versions of a PDF page.

Potential code breaking changes:

- The private method `Page._getTransformation()` has been removed. Use the public `Page.transformationMatrix` instead.
-

Changes in Version 1.16.18

This version introduces several new features around PDF text output. The motivation is to simplify this task, while at the same time offering extending features.

One major achievement is using MuPDF’s capabilities to dynamically choosing fallback fonts whenever a character cannot be found in the current one. This seamlessly works for Base-14 fonts in combination with CJK fonts (China, Japan, Korea). So a text may contain **any combination of characters** from the Latin, Greek, Russian, Chinese, Japanese and Korean languages.

- **Fixed** issue #493. `Pixmap(doc, xref)` should now again correctly resemble the loaded image object.
 - **Fixed** issue #488. Widget names are now modifiable.
 - **Added** new class *Font* which represents a font.
 - **Added** new class *TextWriter* which serves as a container for text to be written on a page.
 - **Added** `Page.writeText()` to write one or more *TextWriter* objects to the page.
-

Changes in Version 1.16.17

- **Fixed** issue #479. PyMuPDF should now more correctly report image resolutions. This applies to both, images (either from images files or extracted from PDF documents) and pixmaps created from images.
 - **Added** `Pixmap.set_dpi()` which sets the image resolution in x and y directions.
-

Changes in Version 1.16.16

- **Fixed** issue #477.
 - **Fixed** issue #476.
 - **Changed** annotation line end symbol coloring and fixed an error coloring the interior of ‘Polyline’ / ‘Polygon’ annotations.
-

Changes in Version 1.16.14

- **Changed** text marker annotations to accept parameters beyond just quadrilaterals such that now **text lines between two given points can be marked**.
-

- Added `Document.scrub()` which **removes potentially sensitive data** from a PDF. Implements #453.
 - Added `Annot.blendMode()` which returns the **blend mode** of annotations.
 - Added `Annot.setBlendMode()` to set the annotation's blend mode. This resolves issue #416.
 - Changed `Annot.update()` to accept additional parameters for setting blend mode and opacity.
 - Added advanced graphics features to **control the anti-aliasing values**, `Tools.set_aa_level()`. Resolves #467
 - Fixed issue #474.
 - Fixed issue #466.
-

Changes in Version 1.16.13

- Added `Document.getPageXObjectList()` which returns a list of **Form XObjects** of the page.
 - Added `Page.setMediaBox()` for changing the physical PDF page size.
 - Added `Page` methods which have been internal before: `Page.cleanContents()` (= `Page._cleanContents()`), `Page.getContents()` (= `Page._getContents()`), `Page.getTransformation()` (= `Page._getTransformation()`).
-

Changes in Version 1.16.12

- Fixed issue #447
 - Fixed issue #461.
 - Fixed issue #397.
 - Fixed issue #463.
 - Added JavaScript support to PDF form fields, thereby fixing #454.
 - Added a new annotation method `Annot.delete_responses()`, which removes 'Popup' and response annotations referring to the current one. Mainly serves data protection purposes.
 - Added a new form field method `Widget.reset()`, which resets the field value to its default.
 - Changed and extended handling of redactions: images and XObjects are removed if *contained* in a redaction rectangle. Any partial only overlaps will just be covered by the redaction background color. Now an *overlay* text can be specified to be inserted in the rectangle area to **take the place the deleted original** text. This resolves #434.
-

Changes in Version 1.16.11

- Added Support for redaction annotations via method `Page.add_redact_annot()` and `Page.apply_redactions()`.
 - Fixed issue #426 ("PolygonAnnotation in 1.16.10 version").
 - Fixed documentation only issues #443 and #444.
-

Changes in Version 1.16.10

- Fixed issue #421 ("annot.set_rect(rect) has no effect on text Annotation")
 - Fixed issue #417 ("Strange behavior for page.deleteAnnot on 1.16.9 compare to 1.13.20")
-

- **Fixed** issue #415 (“Annot.setOpacity throws mupdf warnings”)
 - **Changed** all “add annotation / widget” methods to store a unique name in the */NM* PDF key.
 - **Changed** `Annot.setInfo()` to also accept direct parameters in addition to a dictionary.
 - **Changed** `Annot.info` to now also show the annotation’s unique id (*/NM* PDF key) if present.
 - **Added** `Page.annot_names()` which returns a list of all annotation names (*/NM* keys).
 - **Added** `Page.load_annot()` which loads an annotation given its unique id (*/NM* key).
 - **Added** `Document.reload_page()` which provides a new copy of a page after finishing any pending updates to it.
-

Changes in Version 1.16.9

- **Fixed** #412 (“Feature Request: Allow controlling whether TOC entries should be collapsed”)
 - **Fixed** #411 (“Seg Fault with page.firstWidget”)
 - **Fixed** #407 (“Annot.setOpacity trouble”)
 - **Changed** methods `Annot.setBorder()`, `Annot.setColors()`, `Link.setBorder()`, and `Link.setColors()` to also accept direct parameters, and not just cumbersome dictionaries.
-

Changes in Version 1.16.8

- **Added** several new methods to the `Document` class, which make dealing with PDF low-level structures easier. I also decided to provide them as “normal” methods (as opposed to private ones starting with an underscore “_”). These are `Document.xrefObject()`, `Document.xrefStream()`, `Document.xrefStreamRaw()`, `Document.PDFTrailer()`, `Document.PDFCatalog()`, `Document.metadataXML()`, `Document.updateObject()`, `Document.updateStream()`.
 - **Added** `Tools.mupdf_disply_errors()` which sets the display of mupdf errors on `sys.stderr`.
 - **Added** a commandline facility. This a major new feature: you can now invoke several utility functions via “`python -m fitz ...`”. It should obsolete the need for many of the most trivial scripts. Please refer to [Module fitz](#).
-

Changes in Version 1.16.7

Minor changes to better synchronize the binary image streams of `TextPage` image blocks and `Document.extractImage()` images.

- **Fixed** issue #394 (“PyMuPDF Segfaults when using `TOOLS.mupdf_warnings()`”).
 - **Changed** redirection of MuPDF error messages: apart from writing them to Python `sys.stderr`, they are now also stored with the MuPDF warnings.
 - **Changed** `Tools.mupdf_warnings()` to automatically empty the store (if not deactivated via a parameter).
 - **Changed** `Page.getImageBbox()` to return an **infinite rectangle** if the image could not be located on the page – instead of raising an exception.
-

Changes in Version 1.16.6

- **Fixed** issue #390 (“Incomplete deletion of annotations”).
 - **Changed** `Page.searchFor()` / `Document.searchPageFor()` to also support the *flags* parameter, which controls the data included in a `TextPage`.
-

- **Changed** `Document.getPageImageList()`, `Document.getPageFontList()` and their *Page* counterparts to support a new parameter *full*. If true, the returned items will contain the *xref* of the *FormXObject* where the font or image is referenced.
-

Changes in Version 1.16.5

More performance improvements for text extraction.

- **Fixed** second part of issue #381 (see item in v1.16.4).
 - **Added** `Page.getTextPage()`, so it is no longer required to create an intermediate display list for text extractions. Page level wrappers for text extraction and text searching are now based on this, which should improve performance by ca. 5%.
-

Changes in Version 1.16.4

- **Fixed** issue #381 (“`TextPage.extractDICT ... failed ... after upgrading ... to 1.16.3`”)
 - **Added** method `Document.pages()` which delivers a generator iterator over a page range.
 - **Added** method `Page.links()` which delivers a generator iterator over the links of a page.
 - **Added** method `Page.annots()` which delivers a generator iterator over the annotations of a page.
 - **Added** method `Page.widgets()` which delivers a generator iterator over the form fields of a page.
 - **Changed** `Document.is_form_pdf` to now contain the number of widgets, and *False* if not a PDF or this number is zero.
-

Changes in Version 1.16.3

Minor changes compared to version 1.16.2. The code of the “dict” and “rawdict” variants of `Page.getText()` has been ported to C which has greatly improved their performance. This improvement is mostly noticeable with text-oriented documents, where they now should execute almost two times faster.

- **Fixed** issue #369 (“mupdf: cmsCreateTransform failed”) by removing ICC colorspace support.
 - **Changed** `Page.getText()` to accept additional keywords “blocks” and “words”. These will deliver the results of `Page.getTextBlocks()` and `Page.getTextWords()`, respectively. So all text extraction methods are now available via a uniform API. Correspondingly, there are now new methods `TextPage.extractBLOCKS()` and `TextPage.extractWords()`.
 - **Changed** `Page.getText()` to default bit indicator `TEXT_INHIBIT_SPACES` to **off**. Insertion of additional spaces is **not suppressed** by default.
-

Changes in Version 1.16.2

- **Changed** text extraction methods of *Page* to allow detail control of the amount of extracted data.
 - **Added** `planish_line()` which maps a given line (defined as a pair of points) to the x-axis.
 - **Fixed** an issue (w/o Github number) which brought down the interpreter when encountering certain non-UTF-8 encodable characters while using `Page.getText()` with the “dict” option.
 - **Fixed** issue #362 (“Memory Leak with `getText(‘rawDICT’)`”).
-

Changes in Version 1.16.1

- **Added** property `Quad.is_convex` which checks whether a line is contained in the quad if it connects two points of it.
 - **Changed** `Document.insert_pdf()` to now allow dropping or including links and annotations independently during the copy. Fixes issue #352 (“Corrupt PDF data and ...”), which seemed to intermittently occur when using the method for some problematic PDF files.
 - **Fixed** a bug which, in matrix division using the syntax “`m1/m2`”, caused matrix “`m1`” to be **replaced** by the result instead of delivering a new matrix.
 - **Fixed** issue #354 (“SyntaxWarning with Python 3.8”). We now always use “`==`” for literals (instead of the “`is`” Python keyword).
 - **Fixed** issue #353 (“mupdf version check”), to no longer refuse the import when there are only patch level deviations from MuPDF.
-

Changes in Version 1.16.0

This major new version of MuPDF comes with several nice new or changed features. Some of them imply programming API changes, however. This is a synopsis of what has changed:

- PDF document encryption and decryption is now **fully supported**. This includes setting **permissions**, **passwords** (user and owner passwords) and the desired encryption method.
- In response to the new encryption features, PyMuPDF returns an integer (ie. a combination of bits) for document permissions, and no longer a dictionary.
- Redirection of MuPDF errors and warnings is now natively supported. PyMuPDF redirects error messages from MuPDF to `sys.stderr` and no longer buffers them. Warnings continue to be buffered and will not be displayed. Functions exist to access and reset the warnings buffer.
- Annotations are now **only supported for PDF**.
- Annotations and widgets (form fields) are now **separate object chains** on a page (although widgets technically still **are** PDF annotations). This means, that you will **never encounter widgets** when using `Page.firstAnnot` or `Annot.next()`. You must use `Page.firstWidget` and `Widget.next()` to access form fields.
- As part of MuPDF’s changes regarding widgets, only the following four fonts are supported, when **adding** or **changing** form fields: **Courier**, **Helvetica**, **Times-Roman** and **ZapfDingBats**.

List of change details:

- **Added** `Document.can_save_incrementally()` which checks conditions that are preventing use of option `incremental=True` of `Document.save()`.
- **Added** `Page.firstWidget` which points to the first field on a page.
- **Added** `Page.getImageBbox()` which returns the rectangle occupied by an image shown on the page.
- **Added** `Annot.setName()` which lets you change the (icon) name field.
- **Added** outputting the text color in `Page.getText()`: the “`dict`”, “`rawdict`” and “`xml`” options now also show the color in sRGB format.
- **Changed** `Document.permissions` to now contain an integer of bool indicators – was a dictionary before.
- **Changed** `Document.save()`, `Document.write()`, which now fully support password-based decryption and encryption of PDF files.
- **Changed the names of all Python constants** related to annotations and widgets. Please make sure to consult the **Constants and Enumerations** chapter if your script is dealing with these two classes. This decision goes back to the dropped support for non-PDF annotations. The **old names** (starting with “`ANNOT_*`” or “`WIDGET_*`”) will be available as deprecated synonyms.

- **Changed** font support for widgets: only *Cour* (Courier), *Helv* (Helvetica, default), *TiRo* (Times-Roman) and *ZaDb* (ZapfDingBats) are accepted when **adding or changing** form fields. Only the plain versions are possible – not their italic or bold variations. **Reading** widgets, however will show its original font.
- **Changed** the name of the warnings buffer to `Tools.mupdf_warnings()` and the function to empty this buffer is now called `Tools.reset_mupdf_warnings()`.
- **Changed** `Page.getPixmap()`, `Document.get_page_pixmap()`: a new bool argument *annots* can now be used to **suppress the rendering of annotations** on the page.
- **Changed** `Page.add_file_annot()` and `Page.add_text_annot()` to enable setting an icon.
- **Removed** widget-related methods and attributes from the *Annot* object.
- **Removed** *Document* attributes *openErrCode*, *openErrMsg*, and *Tools* attributes / methods *stderr*, *reset_stderr*, *stdout*, and *reset_stdout*.
- **Removed** **thirdparty zlib** dependency in PyMuPDF: there are now compression functions available in MuPDF. Source installers of PyMuPDF may now omit this extra installation step.

No version published for MuPDF v1.15.0

Changes in Version 1.14.20 / 1.14.21

- **Changed** text marker annotations to support multiple rectangles / quadrilaterals. This fixes issue #341 (“Question : How to addhighlight so that a string spread across more than a line is covered by one highlight?”) and similar (#285).
 - **Fixed** issue #331 (“Importing PyMuPDF changes warning filtering behaviour globally”).
-

Changes in Version 1.14.19

- **Fixed** issue #319 (“InsertText function error when use custom font”).
 - **Added** new method `Document.get_sigflags()` which returns information on whether a PDF is signed. Resolves issue #326 (“How to detect signature in a form pdf?”).
-

Changes in Version 1.14.17

- **Added** `Document.fullcopyPage()` to make full page copies within a PDF (not just copied references as `Document.copyPage()` does).
 - **Changed** `Page.getPixmap()`, `Document.get_page_pixmap()` now use *alpha=False* as default.
 - **Changed** text extraction: the span dictionary now (again) contains its rectangle under the *bbox* key.
 - **Changed** `Document.movePage()` and `Document.copyPage()` to use direct functions instead of wrapping `Document.select()` – similar to `Document.delete_page()` in v1.14.16.
-

Changes in Version 1.14.16

- **Changed** *Document* methods around PDF */EmbeddedFiles* to no longer use MuPDF’s “portfolio” functions. That support will be dropped in MuPDF v1.15 – therefore another solution was required.
- **Changed** `Document.embfile_Count()` to be a function (was an attribute).
- **Added** new method `Document.embfile_Names()` which returns a list of names of embedded files.

- **Changed** `Document.delete_page()` and `Document.delete_pages()` to internally no longer use `Document.select()`, but instead use functions to perform the deletion directly. As it has turned out, the `Document.select()` method yields invalid outline trees (tables of content) for very complex PDFs and sophisticated use of annotations.
-

Changes in Version 1.14.15

- **Fixed** issues #301 (“Line cap and Line join”), #300 (“How to draw a shape without outlines”) and #298 (“utils.updateRect exception”). These bugs pertain to drawing shapes with PyMuPDF. Drawing shapes without any border is fully supported. Line cap styles and line line join style are now differentiated and support all possible PDF values (0, 1, 2) instead of just being a bool. The previous parameter `roundCap` is deprecated in favor of `lineCap` and `lineJoin` and will be deleted in the next release.
 - **Fixed** issue #290 (“Memory Leak with getText(‘rawDICT’)”). This bug caused memory not being (completely) freed after invoking the “dict”, “rawdict” and “json” versions of `Page.getText()`.
-

Changes in Version 1.14.14

- **Added** new low-level function `ImageProperties()` to determine a number of characteristics for an image.
 - **Added** new low-level function `Document.is_stream()`, which checks whether an object is of stream type.
 - **Changed** low-level functions `Document._getXrefString()` and `Document._getTrailerString()` now by default return object definitions in a formatted form which makes parsing easy.
-

Changes in Version 1.14.13

- **Changed** methods working with binary input: while ever supporting bytes and bytearray objects, they now also accept `io.BytesIO` input, using their `getvalue()` method. This pertains to document creation, embedded files, FileAttachment annotations, pixmap creation and others. Fixes issue #274 (“Segfault when using BytesIO as a stream for insertImage”).
 - **Fixed** issue #278 (“Is insertImage(keep_proportion=True) broken?”). Images are now correctly presented when keeping aspect ratio.
-

Changes in Version 1.14.12

- **Changed** the draw methods of `Page` and `Shape` to support not only RGB, but also GRAY and CMYK colorspaces. This solves issue #270 (“Is there a way to use CMYK color to draw shapes?”). This change also applies to text insertion methods of `Shape`, resp. `Page`.
 - **Fixed** issue #269 (“AttributeError in Document.insert_page()”), which occurred when using `Document.insert_page()` with text insertion.
-

Changes in Version 1.14.11

- **Changed** `Page.show_pdf_page()` to always position the source rectangle centered in the target. This method now also supports **rotation by arbitrary angles**. The argument `reuse_xref` has been deprecated: prevention of duplicates is now **handled internally**.
 - **Changed** `Page.insertImage()` to support rotated display of the image and keeping the aspect ratio. Only rotations by multiples of 90 degrees are supported here.
-

- **Fixed** issue #265 (“TypeError: insertText() got an unexpected keyword argument ‘idx’”). This issue only occurred when using `Document.insert_page()` with also inserting text.
-

Changes in Version 1.14.10

- **Changed** `Page.show_pdf_page()` to support rotation of the source rectangle. Fixes #261 (“Cannot rotate insterted pages”).
 - **Fixed** a bug in `Page.insertImage()` which prevented insertion of multiple images provided as streams.
-

Changes in Version 1.14.9

- **Added** new low-level method `Document._getTrailerString()`, which returns the trailer object of a PDF. This is much like `Document._getXrefString()` except that the PDF trailer has no / needs no *xref* to identify it.
 - **Added** new parameters for text insertion methods. You can now set stroke and fill colors of glyphs (text characters) independently, as well as the thickness of the glyph border. A new parameter *render_mode* controls the use of these colors, and whether the text should be visible at all.
 - **Fixed** issue #258 (“Copying image streams to new PDF without size increase”): For JPX images embedded in a PDF, `Document.extractImage()` will now return them in their original format. Previously, the MuPDF base library was used, which returns them in PNG format (entailing a massive size increase).
 - **Fixed** issue #259 (“Morphing text to fit inside rect”). Clarified use of `get_text_length()` and removed extra line breaks for long words.
-

Changes in Version 1.14.8

- **Added** `Pixmap.set_rect()` to change the pixel values in a rectangle. This is also an alternative to setting the color of a complete pixmap (`Pixmap.clear_with()`).
 - **Fixed** an image extraction issue with JBIG2 (monochrome) encoded PDF images. The issue occurred in `Page.getText()` (parameters “dict” and “rawdict”) and in `Document.extractImage()` methods.
 - **Fixed** an issue with not correctly clearing a non-alpha *Pixmap* (`Pixmap.clear_with()`).
 - **Fixed** an issue with not correctly inverting colors of a non-alpha *Pixmap* (`Pixmap.invert_irect()`).
-

Changes in Version 1.14.7

- **Added** `Pixmap.set_pixel()` to change one pixel value.
 - **Added** documentation for image conversion in the FAQ.
 - **Added** new function `get_text_length()` to determine the string length for a given font.
 - **Added** Postscript image output (changed `Pixmap.save()` and `Pixmap.tobytes()`).
 - **Changed** `Pixmap.save()` and `Pixmap.tobytes()` to ensure valid combinations of colorspace, alpha and output format.
 - **Changed** `Pixmap.save()`: the desired format is now inferred from the filename.
 - **Changed** FreeText annotations can now have a transparent background - see `Annot.update()`.
-

Changes in Version 1.14.5

- **Changed:** *Shape* methods now strictly use the transformation matrix of the *Page* – instead of “manually” calculating locations.
 - **Added** method *Pixmap.pixel()* which returns the pixel value (a list) for given pixel coordinates.
 - **Added** method *Pixmap.tobytes()* which returns a bytes object representing the pixmap in a variety of formats. Previously, this could be done for PNG outputs only (*Pixmap.tobytes()*).
 - **Changed:** output of methods *Pixmap.save()* and (the new) *Pixmap.tobytes()* may now also be PSD (Adobe Photoshop Document).
 - **Added** method *Shape.drawQuad()* which draws a *Quad*. This actually is a shorthand for a *Shape.drawPolyline()* with the edges of the quad.
 - **Changed** method *Shape.drawOval()*: the argument can now be **either** a rectangle (*rect_like*) **or** a quadrilateral (*quad_like*).
-

Changes in Version 1.14.4

- **Fixes** issue #239 “Annotation coordinate consistency”.
-

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.
-

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.
-

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.show_pdf_page(rect, ...)”` parameter *rect* may now be any *rect_like* sequence.

We also invested considerable effort to further extend and improve the FAQ chapter.

Changes in Version 1.13.19

This version contains some technical / performance improvements and bug fixes.

- **Changed** memory management: for Python 3 builds, Python memory management is exclusively used across all C-level code (i.e. no more native *malloc()* in MuPDF code or PyMuPDF interface code). This leads to improved memory usage profiles and also some runtime improvements: we have seen > 2% shorter runtimes for text extractions and pixmap creations (on Windows machines only to date).
 - **Fixed** an error occurring in Python 2.7, which crashed the interpreter when using `TextPage.extractRAWDICT()` (= `Page.getText(“rawdict”)`).
 - **Fixed** an error occurring in Python 2.7, when creating link destinations.
 - **Extended** the FAQ chapter with more examples.
-

Changes in Version 1.13.18

- **Added** method `TextPage.extractRAWDICT()`, and a corresponding new string parameter “rawdict” to method `Page.getText()`. It extracts text and images from a page in Python *dict* form like `TextPage.extractDICT()`, but with the detail level of `TextPage.extractXML()`, which is position information down to each single character.
-

Changes in Version 1.13.17

- **Fixed** an error that intermittently caused an exception in `Page.show_pdf_page()`, 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 `paper_rect()` which returns a *Rect* for a supplied paper format string. Example: `fitz.paper_rect("letter") = fitz.Rect(0.0, 0.0, 612.0, 792.0)`.
 - **Added** a FAQ chapter to this document.
-

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.
-

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.
-

Changes in Version 1.13.14

This patch version contains several improvements, mainly for annotations.

- **Changed** `Annot.lineEnds` is now a list of two integers representing the line end symbols. Previously was a *dict* of strings.
 - **Added** support of line end symbols for applicable annotations. PyMuPDF now can generate these annotations including the line end symbols.
 - **Added** `Annot.setLineEnds()` adds line end symbols to applicable annotation types ('Line', 'PolyLine', 'Polygon').
 - **Changed** technical implementation of `Page.insertImage()` and `Page.show_pdf_page()`: 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.
-

Changes in Version 1.13.13

This patch version contains several improvements for embedded files and file attachment annotations.

- **Added** `Document.embfile_Upd()` which allows changing **file content and metadata** of an embedded file. It supersedes the old method `Document.embfile_SetInfo()` (which will be deleted in a future version). Content is automatically compressed and metadata may be unicode.
-

- **Changed** `Document.embfile_Add()` to now automatically compress file content. Accompanying metadata can now be unicode (had to be ASCII in the past).
- **Changed** `Document.embfile_Del()` 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.add_file_annot()` 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.

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.

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”).

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()`.
-

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.
-

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.
-

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.
-

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.
-

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.is_form_pdf` 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).
-

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.
-

Changes in Version 1.13.0

This version is based on MuPDF v1.13.0. This release is “primarily a bug fix release”.

In PyMuPDF, we are also doing some bug fixes while introducing minor enhancements. There only very minimal changes to the user’s API.

- `Document` construction is more flexible: the new `filetype` parameter allows setting the document type. If specified, any extension in the filename will be ignored. More completely addresses [issue #156](#). As part of this, the documentation has been reworked.
 - **Changes to `Pixmap` constructors:**
 - Colorspace conversion no longer allows dropping the alpha channel: source and target **alpha will now always be the same**. We have seen exceptions and even interpreter crashes when using `alpha = 0`.
 - As a replacement, the simple pixmap copy lets you choose the target alpha.
 - `Document.save()` again offers the full garbage collection range 0 thru 4. Because of a bug in `xref` maintenance, we had to temporarily enforce `garbage > 1`. Finally resolves [issue #148](#).
 - `Document.save()` now offers to “prettify” PDF source via an additional argument.
 - `Page.insertImage()` has the additional `stream` -parameter, specifying a memory area holding an image.
 - Issue with garbled PNGs on Linux systems has been resolved (“[Problem writing PNG](#)” #133).
-

Changes in Version 1.12.4

This is an extension of 1.12.3.

- Fix of [issue #147](#): methods `Document.getPageFontlist()` and `Document.getPageImagelist()` now also show fonts and images contained in [resources](#) nested via “Form XObjects”.
 - Temporary fix of [issue #148](#): Saving to new PDF files will now automatically use `garbage = 2` if a lower value is given. Final fix is to be expected with MuPDF’s next version. At that point we will remove this circumvention.
 - Preventive fix of illegally using stencil / image mask 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.
-

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.save()* and many others).
-

Changes in Version 1.12.2

This is an extension of 1.12.1.

- Method *Page.show_pdf_page()* 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.
-

Changes in Version 1.12.1

This is an extension of version 1.12.0.

- New method *Page.show_pdf_page()* 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.
-

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.get_toc()* 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.

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.

Changes in Version 1.11.1

This is an extension of v1.11.0.

- New class *Shape*. It facilitates and extends the creation of image shapes on PDF pages. It contains multiple methods for creating elementary shapes like lines, rectangles or circles, which can be combined into more complex ones and be given common properties like line width or colors. Combined shapes are handled as a unit and e.g. be “morphed” together. The class can accumulate multiple complex shapes and put them all in the page's foreground or background – thus also reducing the number of updates to the page's *contents* object.
- All *Page* draw methods now use the new *Shape* class.
- Text insertion methods `insertText()` and `insertTextBox()` now support morphing in addition to text rotation. They have become part of the *Shape* class and thus allow text to be freely combined with graphics.
- A new *Pixmap* constructor allows creating pixmap copies with an added alpha channel. A new method also allows directly manipulating alpha values.
- Binary algebraic operations with geometry objects (matrices, rectangles and points) now generally also support lists or tuples as the second operand. You can add a tuple (x, y) of numbers to a *Point*. In this context, such sequences are called “*point_like*” (resp. *matrix_like*, *rect_like*).
- Geometry objects now fully support in-place operators. For example, $p /= m$ replaces point *p* with $p * 1/m$ for a number, or $p * \sim m$ for a *matrix_like* object *m*. Similarly, if *r* is a rectangle, then $r |= (3, 4)$ is the new rectangle that also includes `fitz.Point(3, 4)`, and $r \&= (1, 2, 3, 4)$ is its intersection with `fitz.Rect(1, 2, 3, 4)`.

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:
 - *embfile_Info()* 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).
 - *embfile_Get()* retrieves the (decompressed) content of an entry into a *bytes* buffer.
 - *embfile_Add(...)* inserts new content into the PDF portfolio. We (in contrast to *mutool*) **restrict** this to entries with a **new name** (no duplicate names allowed).
 - *embfile_Del(...)* deletes an entry from the portfolio (function not offered in MuPDF).
 - *embfile_SetInfo()* – changes filename or description of an embedded file.
 - *embfile_Count* – 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.insert_page()* 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.

Changes in Version 1.10.0

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.

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.insert_pdf()* 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 $n < \text{pageCount}$. So, e.g. *doc[-500]* is always valid and will load page $(-500) \% \text{pageCount}$.
- 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.

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.
 - *delete_page()* deletes a page
 - *delete_pages()* 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.
-

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.insert_pdf()* (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 color space converted to *cspace*. This works for all color space combinations.
 - Pixmap constructor `fitz.Pixmap(colorspace, width, height, samples)` now allows *samples* to also be *bytes*, not only *bytearray*.
-

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 `get_area()` for both *fitz.Rect* and *fitz.IRect*
 - Pixmapes 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 *tobytes()* 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), *include_point()* (enlarge rectangle to also contain a point), *include_rect()* (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 *set_metadata()*. 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 *set_toc(list)*. Supports incremental saves.

DEPRECATED NAMES

The original naming convention for methods and properties has been “camelCase”. Since its creation around 2013, a tremendous increase of functionality has happened in PyMuPDF – and with it a corresponding increase in classes, methods and properties. In too many cases, this has led to non-intuitive, illogical and ugly names, difficult to memorize or guess.

A few versions ago, I therefore decided to shift gears and switch to a “snake_cased” naming standard. This was a major effort, which needed a step-wise approach. I think am done with it now (version 1.18.14).

The following list maps deprecated names to their new versions. For example, property `pageCount` became `page_count` in the *Document* class. There also are less obvious name changes, e.g. method `getPNGdata` was re-named to `tobytes` in the *Pixmap* class.

Names of classes (camel case) and package-wide constants (the majority is upper case) remain untouched.

Old names will remain available as deprecated aliases through MuPDF version 1.19.0 and **be removed** in the version that follows it - probably version 1.20.0, but this depends on upstream decisions (MuPDF).

Starting with version 1.19.0, we will issue deprecation warnings on `sys.stderr` like `Deprecation: 'newPage' removed from class 'Document' after v1.19.0 - use 'new_page'.` when aliased methods are being used. Using a deprecated property will not cause this type of warning.

Starting immediately, all deprecated objects (methods and properties) will show a copy of the original’s docstring, **prefixed** with the deprecation message, for example:

```
>>> print(fitz.Document.pageCount.__doc__)
*** Deprecated and removed in version following 1.19.0 - use 'page_count'. ***
Number of pages.
>>> print(fitz.Document.newPage.__doc__)
*** Deprecated and removed in version following 1.19.0 - use 'new_page'. ***
Create and return a new page object.

    Args:
        pno: (int) insert before this page. Default: after last page.
        width: (float) page width in points. Default: 595 (ISO A4 width).
        height: (float) page height in points. Default 842 (ISO A4 height).
    Returns:
        A Page object.
```

There is a utility script *alias-changer.py* which can be used to do mass-renames in your scripts. It accepts either a single file or a folder as argument. If a folder is supplied, all its Python files and those of its subfolders are changed. Optionally, backups of the scripts can be taken.

Deprecated names are not separately documented. The following list will help you find the documentation of the original.

Note: This is automatically generated. One or two items refer to yet undocumented methods - please simply ignore them.

- `_isWrapped` – *Page.is_wrapped*
- `addCaretAnnot` – *Page.add_caret_annot()*
- `addCircleAnnot` – *Page.add_circle_annot()*
- `addFileAnnot` – *Page.add_file_annot()*
- `addFreetextAnnot` – *Page.add_freetext_annot()*
- `addHighlightAnnot` – *Page.add_highlight_annot()*
- `addInkAnnot` – *Page.add_ink_annot()*
- `addLineAnnot` – *Page.add_line_annot()*
- `addPolygonAnnot` – *Page.add_polygon_annot()*
- `addPolylineAnnot` – *Page.add_polyline_annot()*
- `addRectAnnot` – *Page.add_rect_annot()*
- `addRedactAnnot` – *Page.add_redact_annot()*
- `addSquigglyAnnot` – *Page.add_squiggly_annot()*
- `addStampAnnot` – *Page.add_stamp_annot()*
- `addStrikeoutAnnot` – *Page.add_strikeout_annot()*
- `addTextAnnot` – *Page.add_text_annot()*
- `addUnderlineAnnot` – *Page.add_underline_annot()*
- `addWidget` – *Page.add_widget()*
- `chapterCount` – *Document.chapter_count*
- `chapterPageCount` – *Document.chapter_page_count()*
- `cleanContents` – *Page.clean_contents()*
- `clearWith` – *Pixmap.clear_with()*
- `convertToPDF` – *Document.convert_to_pdf()*
- `copyPage` – *Document.copy_page()*
- `copyPixmap` – *Pixmap.copy()*
- `CropBox` – *Page.cropbox*
- `CropBoxPosition` – *Page.cropbox_position*
- `deleteAnnot` – *Page.delete_annot()*
- `deleteLink` – *Page.delete_link()*
- `deletePage` – *Document.delete_page()*
- `deletePageRange` – *Document.delete_pages()*
- `deleteWidget` – *Page.delete_widget()*
- `derotationMatrix` – *Page.derotation_matrix*

- `drawBezier` – *Page.draw_bezier()*
- `drawBezier` – *Shape.draw_bezier()*
- `drawCircle` – *Page.draw_circle()*
- `drawCircle` – *Shape.draw_circle()*
- `drawCurve` – *Page.draw_curve()*
- `drawCurve` – *Shape.draw_curve()*
- `drawLine` – *Page.draw_line()*
- `drawLine` – *Shape.draw_line()*
- `drawOval` – *Page.draw_oval()*
- `drawOval` – *Shape.draw_oval()*
- `drawPolyline` – *Page.draw_polyline()*
- `drawPolyline` – *Shape.draw_polyline()*
- `drawQuad` – *Page.draw_quad()*
- `drawQuad` – *Shape.draw_quad()*
- `drawRect` – *Page.draw_rect()*
- `drawRect` – *Shape.draw_rect()*
- `drawSector` – *Page.draw_sector()*
- `drawSector` – *Shape.draw_sector()*
- `drawSquiggle` – *Page.draw_squiggle()*
- `drawSquiggle` – *Shape.draw_squiggle()*
- `drawZigzag` – *Page.draw_zigzag()*
- `drawZigzag` – *Shape.draw_zigzag()*
- `embeddedFileAdd` – *Document.embfile_add()*
- `embeddedFileCount` – *Document.embfile_count()*
- `embeddedFileDel` – *Document.embfile_del()*
- `embeddedFileGet` – *Document.embfile_get()*
- `embeddedFileInfo` – *Document.embfile_info()*
- `embeddedFileNames` – *Document.embfile_names()*
- `embeddedFileUpd` – *Document.embfile_upd()*
- `extractFont` – *Document.extract_font()*
- `extractImage` – *Document.extract_image()*
- `fileGet` – *Annot.get_file()*
- `fileUpd` – *Annot.update_file()*
- `fillTextbox` – *TextWriter.fill_textbox()*
- `findBookmark` – *Document.find_bookmark()*
- `firstAnnot` – *Page.first_annot*

- `firstLink` – *Page.first_link*
- `firstWidget` – *Page.first_widget*
- `fullcopyPage` – *Document.fullcopy_page()*
- `gammaWith` – *Pixmap.gamma_with()*
- `getArea` – *Rect.get_area()*
- `getArea` – *IRect.get_area()*
- `getCharWidths` – *Document.get_char_widths()*
- `getContents` – *Page.get_contents()*
- `getDisplayList` – *Page.get_displaylist()*
- `getDrawings` – *Page.get_drawings()*
- `getFontList` – *Page.get_fonts()*
- `getImageBbox` – *Page.get_image_bbox()*
- `getImageData` – *Pixmap.tobytes()*
- `getImageList` – *Page.get_images()*
- `getLinks` – *Page.get_links()*
- `getOCGs` – *Document.get_ocgs()*
- `getPageFontList` – *Document.get_page_fonts()*
- `getPageImageList` – *Document.get_page_images()*
- `getPagePixmap` – *Document.get_page_pixmap()*
- `getPageText` – *Document.get_page_text()*
- `getPageXObjectList` – *Document.get_page_xobjects()*
- `getPDFnow` – *get_pdf_now()*
- `getPDFstr` – *get_pdf_str()*
- `getPixmap` – *Page.get_pixmap()*
- `getPixmap` – *Annot.get_pixmap()*
- `getPixmap` – *DisplayList.get_pixmap()*
- `getPNGData` – *Pixmap.tobytes()*
- `getPNGdata` – *Pixmap.tobytes()*
- `getRectArea` – *Rect.get_area()*
- `getRectArea` – *IRect.get_area()*
- `getSigFlags` – *Document.get_sigflags()*
- `getSVGImage` – *Page.get_svg_image()*
- `getText` – *Page.get_text()*
- `getText` – *Annot.get_text()*
- `getTextBlocks` – *Page.get_text_blocks()*
- `getTextbox` – *Page.get_textbox()*

- `getTextbox` – *Annot.get_textbox()*
- `getLength` – *get_text_length()*
- `getTextPage` – *Page.get_textpage()*
- `getTextPage` – *Annot.get_textpage()*
- `getTextPage` – *DisplayList.get_textpage()*
- `getTextWords` – *Page.get_text_words()*
- `getToC` – *Document.get_toc()*
- `getXmlMetadata` – *Document.get_xml_metadata()*
- `ImageProperties` – *image_properties()*
- `includePoint` – *Rect.include_point()*
- `includePoint` – *IRect.include_point()*
- `includeRect` – *Rect.include_rect()*
- `includeRect` – *IRect.include_rect()*
- `insertFont` – *Page.insert_font()*
- `insertImage` – *Page.insert_image()*
- `insertLink` – *Page.insert_link()*
- `insertPage` – *Document.insert_page()*
- `insertPDF` – *Document.insert_pdf()*
- `insertText` – *Page.insert_text()*
- `insertText` – *Shape.insert_text()*
- `insertTextbox` – *Page.insert_textbox()*
- `insertTextbox` – *Shape.insert_textbox()*
- `invertIRect` – *Pixmap.invert_irect()*
- `isConvex` – *Quad.is_convex*
- `isDirty` – *Document.is_dirty*
- `isEmpty` – *Rect.is_empty*
- `isEmpty` – *IRect.is_empty*
- `isEmpty` – *Quad.is_empty*
- `isFormPDF` – *Document.is_form_pdf*
- `isInfinite` – *Rect.is_infinite*
- `isInfinite` – *IRect.is_infinite*
- `isPDF` – *Document.is_pdf*
- `isRectangular` – *Quad.is_rectangular*
- `isRectilinear` – *Matrix.is_rectilinear*
- `isReflowable` – *Document.is_reflowable*
- `isRepaired` – *Document.is_repaired*

- `isStream` – `Document.is_stream()`
- `lastLocation` – `Document.last_location`
- `lineEnds` – `Annot.line_ends`
- `loadAnnot` – `Page.load_annot()`
- `loadLinks` – `Page.load_links()`
- `loadPage` – `Document.load_page()`
- `makeBookmark` – `Document.make_bookmark()`
- `MediaBox` – `Page.mediabox`
- `MediaBoxSize` – `Page.mediabox_size`
- `metadataXML` – `Document.xref_xml_metadata()`
- `movePage` – `Document.move_page()`
- `needsPass` – `Document.needs_pass`
- `newPage` – `Document.new_page()`
- `newShape` – `Page.new_shape()`
- `nextLocation` – `Document.next_location()`
- `pageCount` – `Document.page_count`
- `pageCropBox` – `Document.page_cropbox()`
- `pageXref` – `Document.page_xref()`
- `PaperRect` – `paper_rect()`
- `PaperSize` – `paper_size()`
- `paperSizes` – `paper_sizes`
- `PDFCatalog` – `Document.pdf_catalog()`
- `PDFTrailer` – `Document.pdf_trailer()`
- `pillowData` – `Pixmap.pil_tobytes()`
- `pillowWrite` – `Pixmap.pil_save()`
- `planishLine` – `planish_line()`
- `preRotate` – `Matrix.prerotate()`
- `preScale` – `Matrix.prescale()`
- `preShear` – `Matrix.preshear()`
- `preTranslate` – `Matrix.pretranslate()`
- `previousLocation` – `Document.prev_location()`
- `readContents` – `Page.read_contents()`
- `resolveLink` – `Document.resolve_link()`
- `rotationMatrix` – `Page.rotation_matrix`
- `searchFor` – `Page.search_for()`
- `searchPageFor` – `Document.search_page_for()`

- `setAlpha` – *Pixmap.set_alpha()*
- `setBlendMode` – *Annot.set_blendmode()*
- `setBorder` – *Annot.set_border()*
- `setColors` – *Annot.set_colors()*
- `setCropBox` – *Page.set_croptbox()*
- `setFlags` – *Annot.set_flags()*
- `setInfo` – *Annot.set_info()*
- `setLanguage` – *Document.set_language()*
- `setLineEnds` – *Annot.set_line_ends()*
- `setMediaBox` – *Page.set_mediabox()*
- `setMetadata` – *Document.set_metadata()*
- `setName` – *Annot.set_name()*
- `setOC` – *Annot.set_oc()*
- `setOpacity` – *Annot.set_opacity()*
- `setOrigin` – *Pixmap.set_origin()*
- `setPixel` – *Pixmap.set_pixel()*
- `setRect` – *Annot.set_rect()*
- `setRect` – *Pixmap.set_rect()*
- `setResolution` – *Pixmap.set_dpi()*
- `setRotation` – *Page.set_rotation()*
- `setToC` – *Document.set_toc()*
- `setXmlMetadata` – *Document.set_xml_metadata()*
- `showPDFpage` – *Page.show_pdf_page()*
- `soundGet` – *Annot.get_sound()*
- `tintWith` – *Pixmap.tint_with()*
- `transformationMatrix` – *Page.transformation_matrix*
- `updateLink` – *Page.update_link()*
- `updateObject` – *Document.update_object()*
- `updateStream` – *Document.update_stream()*
- `wrapContents` – *Page.wrap_contents()*
- `writeImage` – *Pixmap.save()*
- `writePNG` – *Pixmap.save()*
- `writeText` – *Page.write_text()*
- `writeText` – *TextWriter.write_text()*
- `xrefLength` – *Document.xref_length()*
- `xrefObject` – *Document.xref_object()*

- `xrefStream` – *`Document.xref_stream()`*
- `xrefStreamRaw` – *`Document.xref_stream_raw()`*

Symbols

__init__() (Archive method), 148
 __init__() (Colorspace method), 150
 __init__() (Device method), 372
 __init__() (DisplayList method), 151
 __init__() (Document method), 155
 __init__() (DocumentWriter method), 200
 __init__() (IRect method), 209
 __init__() (Matrix method), 218
 __init__() (Pixmap method), 265–267
 __init__() (Point method), 279
 __init__() (Quad method), 281, 282
 __init__() (Rect method), 286
 __init__() (Shape method), 292
 __init__() (Story method), 309
 __init__() (TextWriter method), 324
 _isWrapped, 446

A

a (Matrix attribute), 219
 abs_unit (Point attribute), 280
 add() (Archive method), 149
 add_bullet_list() (Xml method), 339
 add_caret_annot() (Page method), 229
 add_circle_annot() (Page method), 231
 add_class() (Xml method), 342
 add_code() (Xml method), 340
 add_codeblock() (Xml method), 339
 add_description_list() (Xml method), 339
 add_division() (Xml method), 339
 add_file_annot
 examples, 22
 add_file_annot() (Page method), 230
 add_freetext_annot
 align, 230
 border_color, 230
 color, 230
 fill_color, 230
 fontname, 230
 fontsize, 230
 rect, 230
 rotate, 230

 text_color, 230
 add_freetext_annot() (Page method), 230
 add_header() (Xml method), 339
 add_highlight_annot() (Page method), 233
 add_horizontal_line() (Xml method), 339
 add_image() (Xml method), 339
 add_ink_annot() (Page method), 231
 add_kbd() (Xml method), 340
 add_layer() (Document method), 158
 add_line_annot() (Page method), 231
 add_link() (Xml method), 339
 add_number_list() (Xml method), 339
 add_ocg() (Document method), 158
 add_paragraph() (Xml method), 340
 add_polygon_annot() (Page method), 233
 add_polyline_annot() (Page method), 233
 add_rect_annot() (Page method), 231
 add_redact_annot() (Page method), 232
 add_samp() (Xml method), 340
 add_span() (Xml method), 340
 add_squiggly_annot() (Page method), 233
 add_stamp_annot() (Page method), 235
 add_strikeout_annot() (Page method), 233
 add_style() (Xml method), 342
 add_subscript() (Xml method), 340
 add_superscript() (Xml method), 340
 add_text() (Xml method), 340
 add_text_annot() (Page method), 229
 add_underline_annot() (Page method), 233
 add_var() (Xml method), 340
 add_widget() (Page method), 235
 addCaretAnnot, 446
 addCircleAnnot, 446
 addFileAnnot, 446
 addFreetextAnnot, 446
 addHighlightAnnot, 446
 addInkAnnot, 446
 addLineAnnot, 446
 addPolygonAnnot, 446
 addPolylineAnnot, 446
 addRectAnnot, 446
 addRedactAnnot, 446

- addSquigglyAnnot, 446
- addStampAnnot, 446
- addStrikeoutAnnot, 446
- addTextAnnot, 446
- addUnderlineAnnot, 446
- addWidget, 446
- adobe_glyph_names(), 356
- adobe_glyph_unicodes(), 356
- align
 - add_freetext_annot, 230
 - insert_textbox, 239, 300
- alpha
 - Annot.get_pixmap, 136
 - DisplayList.get_pixmap, 152
 - get_pixmap, 253
- alpha (*Pixmap attribute*), 274
- Annot (*built-in class*), 136
- Annot.get_pixmap
 - alpha, 136
 - colorspace, 136
 - dpi, 136
 - matrix, 136
- Annot.get_text
 - blocks, 136
 - clip, 136
 - dict, 136
 - flags, 136
 - html, 136
 - json, 136
 - rawdict, 136
 - text, 136
 - words, 136
 - xhtml, 136
 - xml, 136
- Annot.update
 - blend_mode, 142
 - border_color, 142
 - cross_out, 142
 - fill_color, 142
 - fontsize, 142
 - rotate, 142
 - text_color, 142
- Annot.update_file
 - buffer, 143
 - desc, 143
 - filename, 143
 - ufilename, 143
- annot_names() (*Page method*), 254
- annot_xrefs() (*Page method*), 254
- annots
 - Document.insert_pdf, 181
 - get_pixmap, 253
- annots() (*Page method*), 238
- append() (*TextWriter method*), 324

- append_child() (*Xml method*), 342
- appendv() (*TextWriter method*), 325
- apply_redactions() (*Page method*), 236
- Archive (*built-in class*), 148
- artbox (*Page attribute*), 260
- ascender (*Font attribute*), 207
- attach
 - embed file, 62
- authenticate() (*Document method*), 162

B

- b (*Matrix attribute*), 220
- Base14_Fonts (*built-in variable*), 379
- bbox (*Font attribute*), 207
- begin_page() (*DocumentWriter method*), 200
- bl (*IRect attribute*), 211
- bl (*Rect attribute*), 290
- bleedbox (*Page attribute*), 260
- blend_mode
 - Annot.update, 142
- blendmode (*Annot attribute*), 139
- blocks
 - Annot.get_text, 136
 - Page.get_text, 245
- body (*Story attribute*), 311
- border (*Annot attribute*), 146
- border (*Link attribute*), 214
- border_color
 - add_freetext_annot, 230
 - Annot.update, 142
- border_color (*Widget attribute*), 335
- border_dashes (*Widget attribute*), 335
- border_style (*Widget attribute*), 335
- border_width
 - insert_text, 239, 297
 - insert_textbox, 239, 300
- border_width (*Widget attribute*), 335
- bottom_left (*IRect attribute*), 211
- bottom_left (*Rect attribute*), 290
- bottom_right (*IRect attribute*), 211
- bottom_right (*Rect attribute*), 290
- bound() (*Page method*), 229
- br (*IRect attribute*), 211
- br (*Rect attribute*), 290
- breadth
 - draw_squiggle, 240, 292
 - draw_zigzag, 240, 294
- buffer
 - Annot.update_file, 143
- buffer (*Font attribute*), 206
- button_caption (*Widget attribute*), 335
- button_states() (*Widget method*), 334

C

- `c` (*Matrix attribute*), 220
- `can_save_incrementally()` (*Document method*), 178
- `catalog` (*built-in variable*), 375
- `chapter_count` (*Document attribute*), 197
- `chapter_page_count()` (*Document method*), 164
- `chapterCount`, 446
- `chapterPageCount`, 446
- `char_lengths()` (*Font method*), 206
- `choice_values` (*Widget attribute*), 335
- `clean_contents()` (*Annot method*), 369
- `clean_contents()` (*Page method*), 368
- `cleanContents`, 446
- `clear_with()` (*Pixmap method*), 268
- `clearWith`, 446
- `clip`
 - `Annot.get_text`, 136
 - `DisplayList.get_pixmap`, 152
 - `get_pixmap`, 253
 - `get_textpage`, 247
 - `Page.get_text`, 245
 - `search_for`, 257
 - `show_pdf_page`, 256
- `clone()` (*Xml method*), 343
- `close()` (*Document method*), 187
- `close()` (*DocumentWriter method*), 200
- `closePath`
 - `draw_bezier`, 240
 - `draw_circle`, 240
 - `draw_curve`, 241
 - `draw_line`, 240
 - `draw_oval`, 240
 - `draw_polyline`, 240
 - `draw_quad`, 241
 - `draw_rect`, 241
 - `draw_sector`, 240
 - `draw_squiggle`, 240
 - `draw_zigzag`, 240
 - `finish`, 297
- `color`
 - `add_freertext_annot`, 230
 - `Document.insert_page`, 182
 - `draw_bezier`, 240
 - `draw_circle`, 240
 - `draw_curve`, 241
 - `draw_line`, 240
 - `draw_oval`, 240
 - `draw_polyline`, 240
 - `draw_quad`, 241
 - `draw_rect`, 241
 - `draw_sector`, 240
 - `draw_squiggle`, 240
 - `draw_zigzag`, 240
 - `finish`, 297
 - `insert_text`, 239, 297
 - `insert_textbox`, 239, 300
 - `color` (*TextWriter attribute*), 327
 - `color_count()` (*Pixmap method*), 273
 - `color_topusage()` (*Pixmap method*), 274
 - `colors` (*Annot attribute*), 145
 - `colors` (*Link attribute*), 214
 - `colorspace`
 - `Annot.get_pixmap`, 136
 - `DisplayList.get_pixmap`, 152
 - `get_pixmap`, 253
 - `Colorspace` (*built-in class*), 150
 - `colorspace` (*Pixmap attribute*), 274
 - `commit`
 - `overlay`, 302
 - `commit()` (*Shape method*), 302
 - `concat()` (*Matrix method*), 219
 - `contains()` (*IRect method*), 210
 - `contains()` (*Rect method*), 288
 - `contents` (*built-in variable*), 376
 - `ConversionHeader()`, 363
 - `ConversionTrailer()`, 363
 - `convert_to_pdf`
 - `examples`, 19
 - `convert_to_pdf()` (*Document method*), 167
 - `convertToPDF`, 446
 - `copy`
 - `examples`, 27, 28
 - `copy()` (*Pixmap method*), 270
 - `copy_page()` (*Document method*), 184
 - `copyPage`, 446
 - `copyPixmap`, 446
 - `create_element()` (*Xml method*), 343
 - `create_text_node()` (*Xml method*), 343
 - `CropBox`, 446
 - `CropBox` (*built-in variable*), 375
 - `cropbox` (*Page attribute*), 260
 - `cropbox_position` (*Page attribute*), 260
 - `CropBoxPosition`, 446
 - `cross_out`
 - `Annot.update`, 142
 - `CS_CMYK` (*built-in variable*), 379
 - `CS_GRAY` (*built-in variable*), 379
 - `CS_RGB` (*built-in variable*), 379
 - `csCMYK` (*built-in variable*), 379
 - `csGRAY` (*built-in variable*), 379
 - `csRGB` (*built-in variable*), 379
 - `css_for_pymupdf_font()`, 357
- D**
 - `d` (*Matrix attribute*), 220
 - `dashes`
 - `draw_bezier`, 240
 - `draw_circle`, 240

- draw_curve, 241
- draw_line, 240
- draw_oval, 240
- draw_polyline, 240
- draw_quad, 241
- draw_rect, 241
- draw_sector, 240
- draw_squiggle, 240
- draw_zigzag, 240
- finish, 297
- debug() (*Xml method*), 343
- del_toc_item() (*Document method*), 177
- del_xml_metadata() (*Document method*), 363
- delete
 - pages, 62
- delete_annot() (*Page method*), 235
- delete_image
 - xref, 245
- delete_image() (*Page method*), 245
- delete_link() (*Page method*), 237
- delete_page() (*Document method*), 183
- delete_pages() (*Document method*), 183
- delete_responses() (*Annot method*), 141
- delete_widget() (*Page method*), 236
- deleteAnnot, 446
- deleteLink, 446
- deletePage, 446
- deletePageRange, 446
- deleteWidget, 446
- derotation_matrix (*Page attribute*), 261
- derotationMatrix, 446
- desc
 - Annot.update_file, 143
 - Document.embfile_add, 185
 - Document.embfile_upd, 187
- descender (*Font attribute*), 207
- dest (*Link attribute*), 215
- dest (*linkDest attribute*), 216
- dest (*Outline attribute*), 226
- Device (*built-in class*), 372
- dict
 - Annot.get_text, 136
 - Page.get_text, 245
- dictionary (*built-in variable*), 376
- digest (*Pixmap attribute*), 274
- DisplayList (*built-in class*), 151
- DisplayList.get_pixmap
 - alpha, 152
 - clip, 152
 - colorspace, 152
 - matrix, 152
- distance_to() (*Point method*), 279
- doc (*Shape attribute*), 302
- Document
 - filename, 155
 - filetype, 155
 - fontsize, 155
 - open, 155
 - rect, 155
 - stream, 155
 - Document (*built-in class*), 155
 - Document.convert_to_pdf
 - from_page, 167
 - rotate, 167
 - to_page, 167
 - Document.embfile_add
 - desc, 185
 - filename, 185
 - ufilename, 185
 - Document.embfile_upd
 - desc, 187
 - filename, 187
 - ufilename, 187
 - Document.insert_page
 - color, 182
 - fontfile, 182
 - fontname, 182
 - fontsize, 182
 - height, 182
 - width, 182
 - Document.insert_pdf
 - annots, 181
 - from_page, 181
 - links, 181
 - rotate, 181
 - show_progress, 181
 - start_at, 181
 - to_page, 181
 - Document.layout
 - fontsize, 175
 - height, 175
 - rect, 175
 - width, 175
 - Document.new_page
 - height, 182
 - width, 182
 - DocumentWriter (*built-in class*), 200
 - down (*Outline attribute*), 225
 - dpi
 - Annot.get_pixmap, 136
 - get_pixmap, 253
 - get_textpage_ocr, 247
 - draw() (*Story method*), 311
 - draw_bezier
 - closePath, 240
 - color, 240
 - dashes, 240
 - fill, 240

- fill_opacity, 240
- lineCap, 240
- lineJoin, 240
- morph, 240
- oc, 240
- overlay, 240
- stroke_opacity, 240
- width, 240
- draw_bezier() (*Page method*), 240
- draw_bezier() (*Shape method*), 294
- draw_circle
 - closePath, 240
 - color, 240
 - dashes, 240
 - fill, 240
 - fill_opacity, 240
 - lineCap, 240
 - lineJoin, 240
 - morph, 240
 - oc, 240
 - overlay, 240
 - stroke_opacity, 240
 - width, 240
- draw_circle() (*Page method*), 240
- draw_circle() (*Shape method*), 295
- draw_cont (*Shape attribute*), 302
- draw_curve
 - closePath, 241
 - color, 241
 - dashes, 241
 - fill, 241
 - fill_opacity, 241
 - lineCap, 241
 - lineJoin, 241
 - morph, 241
 - oc, 241
 - overlay, 241
 - stroke_opacity, 241
 - width, 241
- draw_curve() (*Page method*), 241
- draw_curve() (*Shape method*), 296
- draw_line
 - closePath, 240
 - color, 240
 - dashes, 240
 - fill, 240
 - fill_opacity, 240
 - lineCap, 240
 - lineJoin, 240
 - morph, 240
 - oc, 240
 - overlay, 240
 - stroke_opacity, 240
 - width, 240
- draw_line() (*Page method*), 240
- draw_line() (*Shape method*), 292
- draw_oval
 - closePath, 240
 - color, 240
 - dashes, 240
 - fill, 240
 - fill_opacity, 240
 - lineCap, 240
 - lineJoin, 240
 - morph, 240
 - oc, 240
 - overlay, 240
 - stroke_opacity, 240
 - width, 240
- draw_oval() (*Page method*), 240
- draw_oval() (*Shape method*), 295
- draw_polyline
 - closePath, 240
 - color, 240
 - dashes, 240
 - fill, 240
 - fill_opacity, 240
 - lineCap, 240
 - lineJoin, 240
 - morph, 240
 - oc, 240
 - overlay, 240
 - stroke_opacity, 240
 - width, 240
- draw_polyline() (*Page method*), 240
- draw_polyline() (*Shape method*), 294
- draw_quad
 - closePath, 241
 - color, 241
 - dashes, 241
 - fill, 241
 - fill_opacity, 241
 - lineCap, 241
 - lineJoin, 241
 - morph, 241
 - oc, 241
 - overlay, 241
 - stroke_opacity, 241
 - width, 241
- draw_quad() (*Page method*), 241
- draw_quad() (*Shape method*), 297
- draw_rect
 - closePath, 241
 - color, 241
 - dashes, 241
 - fill, 241
 - fill_opacity, 241
 - lineCap, 241

- lineJoin, 241
- morph, 241
- oc, 241
- overlay, 241
- stroke_opacity, 241
- width, 241
- draw_rect() (*Page method*), 241
- draw_rect() (*Shape method*), 297
- draw_sector
 - closePath, 240
 - color, 240
 - dashes, 240
 - fill, 240
 - fill_opacity, 240
 - fullSector, 240, 296
 - lineCap, 240
 - lineJoin, 240
 - morph, 240
 - oc, 240
 - overlay, 240
 - stroke_opacity, 240
 - width, 240
- draw_sector() (*Page method*), 240
- draw_sector() (*Shape method*), 296
- draw_squiggle
 - breadth, 240, 292
 - closePath, 240
 - color, 240
 - dashes, 240
 - fill, 240
 - fill_opacity, 240
 - lineCap, 240
 - lineJoin, 240
 - morph, 240
 - oc, 240
 - overlay, 240
 - stroke_opacity, 240
 - width, 240
- draw_squiggle() (*Page method*), 240
- draw_squiggle() (*Shape method*), 292
- draw_zigzag
 - breadth, 240, 294
 - closePath, 240
 - color, 240
 - dashes, 240
 - fill, 240
 - fill_opacity, 240
 - lineCap, 240
 - lineJoin, 240
 - morph, 240
 - oc, 240
 - overlay, 240
 - stroke_opacity, 240
 - width, 240

- draw_zigzag() (*Page method*), 240
- draw_zigzag() (*Shape method*), 294
- drawBezier, 447
- drawCircle, 447
- drawCurve, 447
- drawLine, 447
- drawOval, 447
- drawPolyline, 447
- drawQuad, 447
- drawRect, 447
- drawSector, 447
- drawSquiggle, 447
- drawZigzag, 447

E

- e (*Matrix attribute*), 220
- element_positions() (*Story method*), 311
- embed
 - file, attach, 62
 - PDF, picture, 22
- embeddedFileAdd, 447
- embeddedFileCount, 447
- embeddedFileDel, 447
- embeddedFileGet, 447
- embeddedFileInfo, 447
- embeddedFileNames, 447
- embeddedFileUpd, 447
- embfile_add
 - examples, 22, 25
- embfile_add() (*Document method*), 185
- embfile_count() (*Document method*), 186
- embfile_del() (*Document method*), 186
- embfile_get() (*Document method*), 186
- embfile_info() (*Document method*), 186
- embfile_names() (*Document method*), 187
- embfile_upd() (*Document method*), 187
- EMPTY_IRECT(), 372
- EMPTY_QUAD(), 372
- EMPTY_RECT(), 372
- encoding
 - insert_font, 241
 - insert_text, 239, 297
 - insert_textbox, 239, 300
- end_page() (*DocumentWriter method*), 200
- entry_list (*Archive attribute*), 150
- even_odd
 - finish, 297
- examples
 - add_file_annot, 22
 - convert_to_pdf, 19
 - copy, 27, 28
 - embfile_add, 22, 25
 - extract_image, 20
 - insert_image, 22, 25

- invert_irect, 28
- JPEG, 25
- PhotoImage, 25
- Photoshop, 25
- Postscript, 25
- save, 25, 28
- set_rect, 28
- show_pdf_page, 22, 25
- tobytes, 25
- expandtabs
 - insert_textbox, 239, 300
- extract
 - image non-PDF, 19
 - image PDF, 20
 - table, 33
 - text rectangle, 32
- extract_font() (*Document method*), 192
- extract_image
 - examples, 20
- extract_image() (*Document method*), 190
- extractBLOCKS() (*TextPage method*), 314
- extractDICT() (*TextPage method*), 315
- extractFont, 447
- extractHTML() (*TextPage method*), 315
- extractImage, 447
- extractJSON() (*TextPage method*), 315
- extractRAWdict() (*TextPage method*), 315
- extractRAWJSON() (*TextPage method*), 316
- extractTEXT() (*TextPage method*), 314
- extractText() (*TextPage method*), 314
- extractWORDS() (*TextPage method*), 315
- extractXHTML() (*TextPage method*), 315
- extractXML() (*TextPage method*), 315
- ez_save() (*Document method*), 180

F

- f (*Matrix attribute*), 220
- field_flags (*Widget attribute*), 335
- field_label (*Widget attribute*), 335
- field_name (*Widget attribute*), 335
- field_type (*Widget attribute*), 335
- field_type_string (*Widget attribute*), 335
- field_value (*Widget attribute*), 335
- file
 - attach embed, 62
- file extension
 - wrong, 61
- file_info (*Annot attribute*), 142
- fileGet, 447
- filename
 - Annot.update_file, 143
 - Document, 155
 - Document.embfile_add, 185
 - Document.embfile_upd, 187

- insert_image, 243
- open, 155
- replace_image, 244
- fileSpec (*linkDest attribute*), 216
- filetype
 - Document, 155
 - open, 155
- fileUpd, 447
- fill
 - draw_bezier, 240
 - draw_circle, 240
 - draw_curve, 241
 - draw_line, 240
 - draw_oval, 240
 - draw_polyline, 240
 - draw_quad, 241
 - draw_rect, 241
 - draw_sector, 240
 - draw_squiggle, 240
 - draw_zigzag, 240
 - finish, 297
 - insert_text, 239, 297
 - insert_textbox, 239, 300
- fill_color
 - add_freetext_annot, 230
 - Annot.update, 142
- fill_color (*Widget attribute*), 335
- fill_opacity
 - draw_bezier, 240
 - draw_circle, 240
 - draw_curve, 241
 - draw_line, 240
 - draw_oval, 240
 - draw_polyline, 240
 - draw_quad, 241
 - draw_rect, 241
 - draw_sector, 240
 - draw_squiggle, 240
 - draw_zigzag, 240
 - finish, 297
 - insert_text, 239, 297
 - insert_textbox, 239
- fill_textbox() (*TextWriter method*), 325
- fillTextbox, 447
- find() (*Xml method*), 343
- find_bookmark() (*Document method*), 164
- find_next() (*Xml method*), 344
- findBookmark, 447
- finish
 - closePath, 297
 - color, 297
 - dashes, 297
 - even_odd, 297
 - fill, 297

- fill_opacity, 297
- lineCap, 297
- lineJoin, 297
- morph, 297
- oc, 297
- stroke_opacity, 297
- width, 297
- finish() (*Shape method*), 297
- first_annot (*Page attribute*), 262
- first_child (*Xml attribute*), 344
- first_link (*Page attribute*), 261
- first_widget (*Page attribute*), 262
- firstAnnot, 447
- firstLink, 448
- firstWidget, 448
- fitz_config (*Tools attribute*), 331
- fitz_fontdescriptors, 359
- flags
 - Annot.get_text, 136
 - get_textpage, 247
 - get_textpage_ocr, 247
 - Page.get_text, 245
 - search_for, 257
- flags (*Annot attribute*), 144
- flags (*Font attribute*), 207
- flags (*Link attribute*), 214
- flags (*linkDest attribute*), 216
- Font (*built-in class*), 201
- fontbuffer
 - insert_font, 241
- fontfile
 - Document.insert_page, 182
 - insert_font, 241
 - insert_text, 239, 297
 - insert_textbox, 239, 300
- fontname
 - add_freetext_annot, 230
 - Document.insert_page, 182
 - insert_font, 241
 - insert_text, 239, 297
 - insert_textbox, 239, 300
- fontsize
 - add_freetext_annot, 230
 - Annot.update, 142
 - Document, 155
 - Document.insert_page, 182
 - Document.layout, 175
 - insert_text, 239, 297
 - insert_textbox, 239, 300
 - open, 155
- FormFonts (*Document attribute*), 197
- from_page
 - Document.convert_to_pdf, 167
 - Document.insert_pdf, 181

- full
 - get_textpage_ocr, 247
- fullcopy_page() (*Document method*), 184
- fullcopyPage, 448
- fullSector
 - draw_sector, 240, 296

G

- gamma_with() (*Pixmap method*), 268
- gammaWith, 448
- gen_id() (*Tools method*), 328
- get_area() (*IRect method*), 209
- get_area() (*Rect method*), 288
- get_attribute_value() (*Xml method*), 340
- get_attributes() (*Xml method*), 340
- get_bboxlog() (*Page method*), 364
- get_cdrawings() (*Page method*), 249
- get_char_widths() (*Document method*), 369
- get_contents() (*Page method*), 368
- get_displaylist() (*Page method*), 368
- get_drawings() (*Page method*), 248
- get_file() (*Annot method*), 143
- get_fonts() (*Page method*), 250
- get_image_bbox
 - transform, 251
- get_image_bbox() (*Page method*), 251
- get_image_info
 - hashes, 250
 - xrefs, 250
- get_image_info() (*Page method*), 250
- get_image_rects
 - transform, 251
- get_image_rects() (*Page method*), 251
- get_images() (*Page method*), 250
- get_label() (*Page method*), 237
- get_layer() (*Document method*), 160
- get_layers() (*Document method*), 157
- get_links() (*Page method*), 238
- get_new_xref() (*Document method*), 370
- get_oc() (*Annot method*), 138
- get_oc() (*Document method*), 157
- get_ocgs() (*Document method*), 160
- get_ocmd() (*Document method*), 159
- get_page_fonts() (*Document method*), 174
- get_page_images() (*Document method*), 173
- get_page_labels() (*Document method*), 163
- get_page_numbers() (*Document method*), 163
- get_page_pixmap() (*Document method*), 172
- get_page_text() (*Document method*), 175
- get_page_xobjects() (*Document method*), 172
- get_pdf_now(), 360
- get_pdf_str(), 361
- get_pixmap
 - alpha, 253

- annots, 253
 - clip, 253
 - colorspace, 253
 - dpi, 253
 - matrix, 253
- get_pixmap() (*Annot method*), 136
- get_pixmap() (*DisplayList method*), 152
- get_pixmap() (*Page method*), 253
- get_sigflags() (*Document method*), 185
- get_sound() (*Annot method*), 143
- get_svg_image
 - matrix, 252
- get_svg_image() (*Page method*), 252
- get_text() (*Annot method*), 136
- get_text() (*Page method*), 245
- get_text_blocks() (*Page method*), 368
- get_text_length(), 361
- get_text_words() (*Page method*), 368
- get_textbox
 - rect, 246
 - textpage, 246
- get_textbox() (*Annot method*), 137
- get_textbox() (*Page method*), 246
- get_textpage
 - clip, 247
 - flags, 247
- get_textpage() (*DisplayList method*), 152
- get_textpage() (*Page method*), 247
- get_textpage_ocr
 - dpi, 247
 - flags, 247
 - full, 247
 - language, 247
- get_textpage_ocr() (*Page method*), 247
- get_texttrace() (*Page method*), 365
- get_toc() (*Document method*), 168
- get_xml_metadata() (*Document method*), 176
- get_xobjects() (*Page method*), 251
- getArea, 448
- getCharWidths, 448
- getContents, 448
- getDisplayList, 448
- getDrawings, 448
- getFontList, 448
- getImageBbox, 448
- getImageData, 448
- getImageList, 448
- getLinks, 448
- getOCGs, 448
- getPageFontList, 448
- getPageImageList, 448
- getPagePixmap, 448
- getPageText, 448
- getPageXObjectList, 448

- getPDFnow, 448
- getPDFstr, 448
- getPixmap, 448
- getPNGData, 448
- getPNGdata, 448
- getRectArea, 448
- getSigFlags, 448
- getSVGimage, 448
- getText, 448
- getTextBlocks, 448
- getTextbox, 448, 449
- getTextLength, 449
- getTextPage, 449
- getTextWords, 449
- getToC, 449
- getXmlMetadata, 449
- glyph_advance() (*Font method*), 204
- glyph_bbox() (*Font method*), 205
- glyph_count (*Font attribute*), 207
- glyph_name_to_unicode(), 355
- glyph_name_to_unicode() (*Font method*), 204

H

- h (*Pixmap attribute*), 276
- has_annots() (*Document method*), 193
- has_entry() (*Archive method*), 149
- has_glyph() (*Font method*), 203
- has_links() (*Document method*), 193
- has_popup (*Annot attribute*), 146
- hashes
 - get_image_info, 250
- height
 - Document.insert_page, 182
 - Document.layout, 175
 - Document.new_page, 182
 - open, 155
- height (*IRect attribute*), 212
- height (*Pixmap attribute*), 276
- height (*Quad attribute*), 283
- height (*Rect attribute*), 290
- height (*Shape attribute*), 302
- html
 - Annot.get_text, 136
 - Page.get_text, 245

I

- image
 - non-PDF, extract, 19
 - PDF, extract, 20
 - resolution, 17
 - SVG, vector, 25
- image_profile(), 362
- ImageProperties, 449
- include_point() (*Rect method*), 287

- `include_rect()` (*Rect method*), 287
- `includePoint`, 449
- `includeRect`, 449
- `INFINITE_Irect()`, 372
- `INFINITE_Quad()`, 372
- `INFINITE_Rect()`, 372
- `info` (*Annot attribute*), 144
- `inheritable` (*built-in variable*), 375
- `insert_after()` (*Xml method*), 343
- `insert_before()` (*Xml method*), 343
- `insert_font`
 - `encoding`, 241
 - `fontbuffer`, 241
 - `fontfile`, 241
 - `fontname`, 241
 - `set_simple`, 241
- `insert_font()` (*Page method*), 241
- `insert_image`
 - `examples`, 22, 25
 - `filename`, 243
 - `keep_proportion`, 243
 - `mask`, 243
 - `oc`, 243
 - `overlay`, 243
 - `pixmap`, 243
 - `rotate`, 243
 - `stream`, 243
 - `xref`, 243
- `insert_image()` (*Page method*), 243
- `insert_link()` (*Page method*), 237
- `insert_page()` (*Document method*), 182
- `insert_pdf()` (*Document method*), 181
- `insert_text`
 - `border_width`, 239, 297
 - `color`, 239, 297
 - `encoding`, 239, 297
 - `fill`, 239, 297
 - `fill_opacity`, 239, 297
 - `fontfile`, 239, 297
 - `fontname`, 239, 297
 - `fontsize`, 239, 297
 - `morph`, 239, 297
 - `oc`, 239, 297
 - `overlay`, 239
 - `render_mode`, 239, 297
 - `rotate`, 239, 297
 - `stroke_opacity`, 239, 297
- `insert_text()` (*Page method*), 239
- `insert_text()` (*Shape method*), 299
- `insert_textbox`
 - `align`, 239, 300
 - `border_width`, 239, 300
 - `color`, 239, 300
 - `encoding`, 239, 300
 - `expandtabs`, 239, 300
 - `fill`, 239, 300
 - `fill_opacity`, 239
 - `fontfile`, 239, 300
 - `fontname`, 239, 300
 - `fontsize`, 239, 300
 - `morph`, 239, 300
 - `oc`, 239, 300
 - `overlay`, 239
 - `render_mode`, 239, 300
 - `rotate`, 239, 300
 - `stroke_opacity`, 239
- `insert_textbox()` (*Page method*), 239
- `insert_textbox()` (*Shape method*), 300
- `insertFont`, 449
- `insertImage`, 449
- `insertLink`, 449
- `insertPage`, 449
- `insertPDF`, 449
- `insertText`, 449
- `insertTextbox`, 449
- `interpolate` (*Pixmap attribute*), 277
- `intersect()` (*Irect method*), 210
- `intersect()` (*Rect method*), 287
- `intersects()` (*Irect method*), 210
- `intersects()` (*Rect method*), 288
- `invert()` (*Matrix method*), 219
- `invert_irect`
 - `examples`, 28
- `invert_irect()` (*Pixmap method*), 270
- `invertIrect`, 449
- `Irect` (*built-in class*), 209
- `irect` (*Pixmap attribute*), 275
- `irect` (*Rect attribute*), 289
- `irect_like` (*built-in variable*), 375
- `irt_xref` (*Annot attribute*), 145
- `is_closed` (*Document attribute*), 195
- `is_convex` (*Quad attribute*), 283
- `is_dirty` (*Document attribute*), 195
- `is_empty` (*Irect attribute*), 212
- `is_empty` (*Quad attribute*), 283
- `is_empty` (*Rect attribute*), 291
- `is_encrypted` (*Document attribute*), 196
- `is_external` (*Outline attribute*), 225
- `is_form_pdf` (*Document attribute*), 195
- `is_infinite` (*Irect attribute*), 212
- `is_infinite` (*Rect attribute*), 290
- `is_monochrome` (*Pixmap attribute*), 275
- `is_open` (*Annot attribute*), 146
- `is_open` (*Outline attribute*), 225
- `is_pdf` (*Document attribute*), 195
- `is_rectangular` (*Quad attribute*), 283
- `is_rectilinear` (*Matrix attribute*), 220
- `is_reflowable` (*Document attribute*), 195

is_repaired (*Document attribute*), 196
 is_signed (*Widget attribute*), 335
 is_stream() (*Document method*), 370
 is_text (*Xml attribute*), 344
 is_unicolor (*Pixmap attribute*), 275
 is_valid (*Rect attribute*), 291
 is_wrapped (*Page attribute*), 368
 is_writable (*Font attribute*), 208
 isConvex, 449
 isDirty, 449
 isEmpty, 449
 isExternal (*Link attribute*), 214
 isFormPDF, 449
 isInfinite, 449
 isMap (*linkDest attribute*), 216
 isPDF, 449
 isRectangular, 449
 isRectilinear, 449
 isReflowable, 449
 isRepaired, 449
 isStream, 450
 isUri (*linkDest attribute*), 216

J

journal_can_do() (*Document method*), 194
 journal_enable() (*Document method*), 193
 journal_load() (*Document method*), 194
 journal_op_name() (*Document method*), 194
 journal_position() (*Document method*), 194
 journal_redo() (*Document method*), 194
 journal_save() (*Document method*), 194
 journal_start_op() (*Document method*), 193
 journal_stop_op() (*Document method*), 194
 journal_undo() (*Document method*), 194
 JPEG
 examples, 25
 json
 Annot.get_text, 136
 Page.get_text, 245

K

keep_proportion
 insert_image, 243
 show_pdf_page, 256
 kind (*linkDest attribute*), 216

L

language
 get_textpage_ocr, 247
 last_child (*Xml attribute*), 344
 last_location (*Document attribute*), 197
 last_point (*TextWriter attribute*), 327
 lastLocation, 450
 lastPoint (*Shape attribute*), 303

layer_ui_configs() (*Document method*), 161
 layout() (*Document method*), 175
 ligature (*built-in variable*), 378
 line_ends (*Annot attribute*), 145
 lineCap
 draw_bezier, 240
 draw_circle, 240
 draw_curve, 241
 draw_line, 240
 draw_oval, 240
 draw_polyline, 240
 draw_quad, 241
 draw_rect, 241
 draw_sector, 240
 draw_squiggle, 240
 draw_zigzag, 240
 finish, 297
 lineEnds, 450
 lineJoin
 draw_bezier, 240
 draw_circle, 240
 draw_curve, 241
 draw_line, 240
 draw_oval, 240
 draw_polyline, 240
 draw_quad, 241
 draw_rect, 241
 draw_sector, 240
 draw_squiggle, 240
 draw_zigzag, 240
 finish, 297
 Link (*built-in class*), 213
 LINK_FLAG_B_VALID (*built-in variable*), 383
 LINK_FLAG_FIT_H (*built-in variable*), 383
 LINK_FLAG_FIT_V (*built-in variable*), 383
 LINK_FLAG_L_VALID (*built-in variable*), 383
 LINK_FLAG_R_IS_ZOOM (*built-in variable*), 383
 LINK_FLAG_R_VALID (*built-in variable*), 383
 LINK_FLAG_T_VALID (*built-in variable*), 383
 LINK_GOTO (*built-in variable*), 382
 LINK_GOTOR (*built-in variable*), 383
 LINK_LAUNCH (*built-in variable*), 383
 LINK_NAMED (*built-in variable*), 383
 LINK_NONE (*built-in variable*), 382
 LINK_URI (*built-in variable*), 382
 linkDest (*built-in class*), 216
 links
 Document.insert_pdf, 181
 links() (*Page method*), 238
 ll (*Quad attribute*), 283
 load_annot() (*Page method*), 255
 load_links() (*Page method*), 255
 load_page() (*Document method*), 165
 load_widget() (*Page method*), 255

loadAnnot, 450
loadLinks, 450
loadPage, 450
lr (*Quad attribute*), 283
lt (*linkDest attribute*), 216

M

make_bookmark() (*Document method*), 164
make_table(), 358
makeBookmark, 450
mask
 insert_image, 243
matrix
 Annot.get_pixmap, 136
 DisplayList.get_pixmap, 152
 get_pixmap, 253
 get_svg_image, 252
Matrix (*built-in class*), 218
matrix_like (*built-in variable*), 375
MediaBox, 450
MediaBox (*built-in variable*), 375
mediabox (*Page attribute*), 261
mediabox_size (*Page attribute*), 261
MediaBoxSize, 450
metadata (*Document attribute*), 196
metadataXML, 450
morph
 draw_bezier, 240
 draw_circle, 240
 draw_curve, 241
 draw_line, 240
 draw_oval, 240
 draw_polyline, 240
 draw_quad, 241
 draw_rect, 241
 draw_sector, 240
 draw_squiggle, 240
 draw_zigzag, 240
 finish, 297
 insert_text, 239, 297
 insert_textbox, 239, 300
morph() (*IRect method*), 210
morph() (*Quad method*), 282
morph() (*Rect method*), 289
move_page() (*Document method*), 184
movePage, 450
mupdf_display_errors() (*Tools method*), 331
mupdf_warnings() (*Tools method*), 331

N

n (*Colorspace attribute*), 150
n (*Pixmap attribute*), 277
name (*Colorspace attribute*), 150
name (*Document attribute*), 197

name (*Font attribute*), 207
named (*linkDest attribute*), 216
need_appearances() (*Document method*), 185
needs_pass (*Document attribute*), 196
needsPass, 450
new_page() (*Document method*), 182
new_shape() (*Page method*), 257
newPage, 450
newShape, 450
newWindow (*linkDest attribute*), 216
next (*Annot attribute*), 144
next (*Link attribute*), 215
next (*Outline attribute*), 225
next (*Widget attribute*), 334
next (*Xml attribute*), 344
next_location() (*Document method*), 165
nextLocation, 450
non-PDF
 extract image, 19
norm() (*IRect method*), 211
norm() (*Matrix method*), 218
norm() (*Point method*), 279
norm() (*Rect method*), 289
normalize() (*IRect method*), 211
normalize() (*Rect method*), 289
number (*Page attribute*), 262

O

object (*built-in variable*), 377
oc
 draw_bezier, 240
 draw_circle, 240
 draw_curve, 241
 draw_line, 240
 draw_oval, 240
 draw_polyline, 240
 draw_quad, 241
 draw_rect, 241
 draw_sector, 240
 draw_squiggle, 240
 draw_zigzag, 240
 finish, 297
 insert_image, 243
 insert_text, 239, 297
 insert_textbox, 239, 300
OCCD (*built-in variable*), 378
OCG (*built-in variable*), 378
OCMD (*built-in variable*), 378
OCPD (*built-in variable*), 378
opacity (*Annot attribute*), 143
opacity (*TextWriter attribute*), 327
open
 Document, 155
 filename, 155

- filetype, 155
- fontsize, 155
- height, 155
- rect, 155
- stream, 155
- width, 155
- Outline (*built-in class*), 225
- outline (*Document attribute*), 195
- outline_xref() (*Document method*), 177
- overlay
 - commit, 302
 - draw_bezier, 240
 - draw_circle, 240
 - draw_curve, 241
 - draw_line, 240
 - draw_oval, 240
 - draw_polyline, 240
 - draw_quad, 241
 - draw_rect, 241
 - draw_sector, 240
 - draw_squiggle, 240
 - draw_zigzag, 240
 - insert_image, 243
 - insert_text, 239
 - insert_textbox, 239
 - show_pdf_page, 256

P

- Page (*built-in class*), 229
- page (*built-in variable*), 377
- page (*linkDest attribute*), 217
- page (*Outline attribute*), 225
- page (*Shape attribute*), 302
- Page.get_text
 - blocks, 245
 - clip, 245
 - dict, 245
 - flags, 245
 - html, 245
 - json, 245
 - rawdict, 245
 - sort, 245
 - text, 245
 - textpage, 245
 - words, 245
 - xhtml, 245
 - xml, 245
- page_count (*Document attribute*), 197
- page_cropbox() (*Document method*), 166
- page_xref() (*Document method*), 166
- pageCount, 450
- pageCropBox, 450
- pages
 - delete, 62
 - rearrange, 62
- pages() (*Document method*), 167
- pagetree (*built-in variable*), 377
- pageXref, 450
- paper_rect(), 355
- paper_size(), 354
- paper_sizes(), 359
- PaperRect, 450
- PaperSize, 450
- paperSizes, 450
- parent (*Annot attribute*), 143
- parent (*Page attribute*), 262
- Partial Pixmaps, 18
- PDF
 - extract image, 20
 - picture embed, 22
- pdf_catalog() (*Document method*), 188
- pdf_trailer() (*Document method*), 188
- PDFCatalog, 450
- pdfcolor, 360
- pdfocr_save() (*Pixmap method*), 271
- pdfocr_tobytes() (*Pixmap method*), 272
- PDFTrailer, 450
- permissions (*Document attribute*), 196
- PhotoImage
 - examples, 25
- Photoshop
 - examples, 25
- picture
 - embed PDF, 22
- pil_save() (*Pixmap method*), 272
- pil_tobytes() (*Pixmap method*), 273
- pillowData, 450
- pillowWrite, 450
- pixel() (*Pixmap method*), 268
- pixmap
 - insert_image, 243
 - replace_image, 244
- Pixmap (*built-in class*), 265
- place() (*Story method*), 310
- planish_line(), 358
- planishLine, 450
- Point (*built-in class*), 279
- point_like (*built-in variable*), 375
- popup_rect (*Annot attribute*), 146
- popup_xref (*Annot attribute*), 145
- Postscript
 - examples, 25
- preRotate, 450
- prerotate() (*Matrix method*), 218
- preScale, 450
- prescale() (*Matrix method*), 219
- preShear, 450
- preshear() (*Matrix method*), 219

preTranslate, 450
pretranslate() (*Matrix method*), 219
prev_location() (*Document method*), 165
previous (*Xml attribute*), 344
previousLocation, 450

Q

Quad (*built-in class*), 281
quad (*IRect attribute*), 211
quad (*Rect attribute*), 290
quad_like (*built-in variable*), 375
quads
 search_for, 257

R

rawdict
 Annot.get_text, 136
 Page.get_text, 245
rb (*linkDest attribute*), 217
read_contents() (*Page method*), 369
read_entry() (*Archive method*), 149
readContents, 450
reading order
 text, 32
rearrange
 pages, 62
recover_char_quad(), 371
recover_line_quad(), 371
recover_quad(), 358, 370
recover_span_quad(), 371
rect
 add_freetext_annot, 230
 Document, 155
 Document.layout, 175
 get_textbox, 246
 open, 155
rect (*Annot attribute*), 144
Rect (*built-in class*), 286
rect (*DisplayList attribute*), 152
rect (*IRect attribute*), 211
rect (*Link attribute*), 214
rect (*Page attribute*), 262
rect (*Quad attribute*), 282
rect (*Shape attribute*), 303
rect (*TextPage attribute*), 317
rect (*TextWriter attribute*), 327
rect (*Widget attribute*), 335
rect_like (*built-in variable*), 375
rectangle
 extract text, 32
reload_page() (*Document method*), 166
remove() (*Xml method*), 343
remove_attribute() (*Xml method*), 340
render_mode

 insert_text, 239, 297
 insert_textbox, 239, 300
replace_image
 filename, 244
 pixmap, 244
 stream, 244
 xref, 244
replace_image() (*Page method*), 244
reset() (*Story method*), 311
reset() (*Widget method*), 334
reset_mupdf_warnings() (*Tools method*), 331
resolution
 image, 17
 zoom, 17
resolution (*built-in variable*), 378
resolveLink, 450
resources (*built-in variable*), 376
root (*Xml attribute*), 344
rotate
 add_freetext_annot, 230
 Annot.update, 142
 Document.convert_to_pdf, 167
 Document.insert_pdf, 181
 insert_image, 243
 insert_text, 239, 297
 insert_textbox, 239, 300
 set_rotation, 255
 show_pdf_page, 256
rotation (*Annot attribute*), 144
rotation (*Page attribute*), 260
rotation_matrix (*Page attribute*), 261
rotationMatrix, 450
round() (*Rect method*), 286
run() (*DisplayList method*), 151
run() (*Page method*), 364

S

samples (*Pixmap attribute*), 275
samples_mv (*Pixmap attribute*), 275
samples_ptr (*Pixmap attribute*), 276
save
 examples, 25, 28
save() (*Document method*), 179
save() (*Pixmap method*), 271
save_snapshot() (*Document method*), 195
saveIncr() (*Document method*), 181
script (*Widget attribute*), 336
script_calc (*Widget attribute*), 336
script_change (*Widget attribute*), 336
script_format (*Widget attribute*), 336
script_stroke (*Widget attribute*), 336
scrub() (*Document method*), 178
search() (*TextPage method*), 316
search_for

clip, 257
 flags, 257
 quads, 257
 textpage, 257
 search_for() (*Page method*), 257
 search_page_for() (*Document method*), 181
 searchFor, 450
 searchPageFor, 450
 select() (*Document method*), 175
 set_aa_level() (*Tools method*), 330
 set_align() (*Xml method*), 340
 set_alpha() (*Pixmap method*), 270
 set_annot_stem() (*Tools method*), 329
 set_artbox() (*Page method*), 260
 set_attribute() (*Xml method*), 340
 set_bgcolor() (*Xml method*), 341
 set_bleedbox() (*Page method*), 260
 set_blendmode() (*Annot method*), 139
 set_bold() (*Xml method*), 341
 set_border() (*Annot method*), 141
 set_border() (*Link method*), 213
 set_color() (*Xml method*), 341
 set_colors() (*Annot method*), 141
 set_colors() (*Link method*), 213
 set_columns() (*Xml method*), 341
 set_contents() (*Page method*), 368
 set_cropbox() (*Page method*), 259
 set_dpi() (*Pixmap method*), 269
 set_flags() (*Annot method*), 141
 set_flags() (*Link method*), 214
 set_font() (*Xml method*), 341
 set_fontsize() (*Xml method*), 341
 set_id() (*Xml method*), 341
 set_info() (*Annot method*), 137
 set_irt_xref() (*Annot method*), 138
 set_italic() (*Xml method*), 341
 set_layer() (*Document method*), 160
 set_layer_ui_config() (*Document method*), 161
 set_leading() (*Xml method*), 341
 set_line_ends() (*Annot method*), 138
 set_lineheight() (*Xml method*), 342
 set_margins() (*Xml method*), 342
 set_mediabox() (*Page method*), 259
 set_metadata() (*Document method*), 175
 set_name() (*Annot method*), 140
 set_oc() (*Annot method*), 138
 set_oc() (*Document method*), 157
 set_ocmd() (*Document method*), 158
 set_opacity() (*Annot method*), 139
 set_open() (*Annot method*), 138
 set_origin() (*Pixmap method*), 269
 set_page_labels() (*Document method*), 163
 set_pagebreak_after() (*Xml method*), 342
 set_pagebreak_before() (*Xml method*), 342
 set_pixel() (*Pixmap method*), 269
 set_popup() (*Annot method*), 139
 set_properties() (*Xml method*), 342
 set_rect
 examples, 28
 set_rect() (*Annot method*), 140
 set_rect() (*Pixmap method*), 269
 set_rotation
 rotate, 255
 set_rotation() (*Annot method*), 140
 set_rotation() (*Page method*), 255
 set_simple
 insert_font, 241
 set_small_glyph_heights() (*Tools method*), 329
 set_subset_fontnames() (*Tools method*), 329
 set_text_indent() (*Xml method*), 342
 set_toc() (*Document method*), 176
 set_toc_item() (*Document method*), 177
 set_trimbox() (*Page method*), 260
 set_xml_metadata() (*Document method*), 176
 setAlpha, 451
 setBlendMode, 451
 setBorder, 451
 setColors, 451
 setCropBox, 451
 setFlags, 451
 setInfo, 451
 setLanguage, 451
 setLineEnds, 451
 setMediaBox, 451
 setMetadata, 451
 setName, 451
 setOC, 451
 setOpacity, 451
 setOrigin, 451
 setPixel, 451
 setRect, 451
 setResolution, 451
 setRotation, 451
 setToC, 451
 setXmlMetadata, 451
 Shape (*built-in class*), 292
 show_aa_level() (*Tools method*), 330
 show_pdf_page
 clip, 256
 examples, 22, 25
 keep_proportion, 256
 overlay, 256
 rotate, 256
 show_pdf_page() (*Page method*), 256
 show_progress
 Document.insert_pdf, 181
 showPDFpage, 451
 shrink() (*Pixmap method*), 268

- size (*Pixmap attribute*), 276
- sort
 - Page.get_text, 245
- soundGet, 451
- sRGB_to_pdf(), 355
- sRGB_to_rgb(), 355
- start_at
 - Document.insert_pdf, 181
- store_maxsize (*Tools attribute*), 333
- store_shrink() (*Tools method*), 330
- store_size (*Tools attribute*), 333
- Story (*built-in class*), 309
- stream
 - Document, 155
 - insert_image, 243
 - open, 155
 - replace_image, 244
- stream (*built-in variable*), 377
- stride (*Pixmap attribute*), 275
- stroke_opacity
 - draw_bezier, 240
 - draw_circle, 240
 - draw_curve, 241
 - draw_line, 240
 - draw_oval, 240
 - draw_polyline, 240
 - draw_quad, 241
 - draw_rect, 241
 - draw_sector, 240
 - draw_squiggle, 240
 - draw_zigzag, 240
 - finish, 297
 - insert_text, 239, 297
 - insert_textbox, 239
- subset_fonts() (*Document method*), 193
- SVG
 - vector image, 25
- switch_layer() (*Document method*), 158

T

- table
 - extract, 33
- tagname (*Xml attribute*), 344
- TESSDATA_PREFIX, 360
- text
 - Annot.get_text, 136
 - Page.get_text, 245
 - reading order, 32
 - rectangle, extract, 32
- text (*Xml attribute*), 344
- TEXT_ALIGN_CENTER (*built-in variable*), 381
- TEXT_ALIGN_JUSTIFY (*built-in variable*), 381
- TEXT_ALIGN_LEFT (*built-in variable*), 381
- TEXT_ALIGN_RIGHT (*built-in variable*), 381

- text_color
 - add_freetext_annot, 230
 - Annot.update, 142
- text_color (*Widget attribute*), 335
- text_cont (*Shape attribute*), 303
- TEXT_DEHYPHENATE (*built-in variable*), 382
- text_font (*Widget attribute*), 335
- text_fontsize (*Widget attribute*), 336
- TEXT_INHIBIT_SPACES (*built-in variable*), 382
- text_length() (*Font method*), 205
- text_maxlen (*Widget attribute*), 336
- TEXT_MEDIABOX_CLIP (*built-in variable*), 382
- TEXT_PRESERVE_IMAGES (*built-in variable*), 381
- TEXT_PRESERVE_LIGATURES (*built-in variable*), 381
- TEXT_PRESERVE_SPANS (*built-in variable*), 382
- TEXT_PRESERVE_WHITESPACE (*built-in variable*), 381
- text_rect (*TextWriter attribute*), 327
- text_type (*Widget attribute*), 336
- TEXTFLAGS_BLOCKS (*built-in variable*), 382
- TEXTFLAGS_DICT (*built-in variable*), 382
- TEXTFLAGS_HTML (*built-in variable*), 382
- TEXTFLAGS_RAWDICT (*built-in variable*), 382
- TEXTFLAGS_SEARCH (*built-in variable*), 382
- TEXTFLAGS_TEXT (*built-in variable*), 382
- TEXTFLAGS_WORDS (*built-in variable*), 382
- TEXTFLAGS_XHTML (*built-in variable*), 382
- TEXTFLAGS_XML (*built-in variable*), 382
- textpage
 - get_textbox, 246
 - Page.get_text, 245
 - search_for, 257
- TextPage (*built-in class*), 314
- TextWriter (*built-in class*), 324
- tint_with() (*Pixmap method*), 268
- tintWith, 451
- title (*Outline attribute*), 225
- tl (*IRect attribute*), 211
- tl (*Rect attribute*), 289
- to_page
 - Document.convert_to_pdf, 167
 - Document.insert_pdf, 181
- tobytes
 - examples, 25
- tobytes() (*Document method*), 181
- tobytes() (*Pixmap method*), 272
- Tools (*built-in class*), 328
- top_left (*IRect attribute*), 211
- top_left (*Rect attribute*), 289
- top_right (*IRect attribute*), 211
- top_right (*Rect attribute*), 289
- torect() (*IRect method*), 210
- torect() (*Rect method*), 288
- totalcont (*Shape attribute*), 303
- tr (*IRect attribute*), 211

tr (*Rect attribute*), 289
 trailer (*built-in variable*), 375
 transform
 get_image_bbox, 251
 get_image_rects, 251
 transform() (*Point method*), 280
 transform() (*Quad method*), 282
 transform() (*Rect method*), 287
 transformation_matrix (*Page attribute*), 261
 transformationMatrix, 451
 trimbox (*Page attribute*), 260
 type (*Annot attribute*), 144

U

ufilename
 Annot.update_file, 143
 Document.embfile_add, 185
 Document.embfile_upd, 187
 ul (*Quad attribute*), 282
 unicode_to_glyph_name(), 356
 unicode_to_glyph_name() (*Font method*), 205
 unit (*Point attribute*), 280
 unitvector (*built-in variable*), 378
 unset_quad_corrections() (*Tools method*), 330
 update() (*Annot method*), 142
 update() (*Widget method*), 334
 update_file() (*Annot method*), 143
 update_link() (*Page method*), 237
 update_object() (*Document method*), 189
 update_stream() (*Document method*), 189
 updateLink, 451
 updateObject, 451
 updateStream, 451
 ur (*Quad attribute*), 283
 uri (*Link attribute*), 214
 uri (*linkDest attribute*), 217
 uri (*Outline attribute*), 225

V

valid_codepoints() (*Font method*), 203
 vector
 image SVG, 25
 version (*built-in variable*), 380
 VersionBind (*built-in variable*), 379
 VersionDate (*built-in variable*), 380
 VersionFitz (*built-in variable*), 380
 vertices (*Annot attribute*), 145

W

w (*Pixmap attribute*), 276
 warp() (*Pixmap method*), 273
 Widget (*built-in class*), 334
 widgets() (*Page method*), 239
 width

Document.insert_page, 182
 Document.layout, 175
 Document.new_page, 182
 draw_bezier, 240
 draw_circle, 240
 draw_curve, 241
 draw_line, 240
 draw_oval, 240
 draw_polyline, 240
 draw_quad, 241
 draw_rect, 241
 draw_sector, 240
 draw_squiggle, 240
 draw_zigzag, 240
 finish, 297
 open, 155
 width (*IRect attribute*), 211
 width (*Pixmap attribute*), 276
 width (*Quad attribute*), 283
 width (*Rect attribute*), 290
 width (*Shape attribute*), 302
 words
 Annot.get_text, 136
 Page.get_text, 245
 wrap_contents() (*Page method*), 367
 wrapContents, 451
 write() (*Story method*), 311
 write_stabilized() (*Story static method*), 312
 write_stabilized_with_links() (*Story static method*), 312
 write_text() (*Page method*), 239
 write_text() (*TextWriter method*), 326
 write_with_links() (*Story method*), 312
 writeImage, 451
 writePNG, 451
 writeText, 451
 wrong
 file extension, 61

X

x (*Pixmap attribute*), 277
 x (*Point attribute*), 280
 x0 (*IRect attribute*), 212
 x0 (*Rect attribute*), 290
 x1 (*IRect attribute*), 212
 x1 (*Rect attribute*), 290
 xhtml
 Annot.get_text, 136
 Page.get_text, 245
 xml
 Annot.get_text, 136
 Page.get_text, 245
 Xml (*built-in class*), 339
 xml_metadata_xref() (*Document method*), 364

- xref
 - delete_image, 245
 - insert_image, 243
 - replace_image, 244
- xref (*Annot attribute*), 145
- xref (*built-in variable*), 378
- xref (*Link attribute*), 215
- xref (*Page attribute*), 262
- xref (*Widget attribute*), 336
- xref_copy() (*Document method*), 190
- xref_get_key() (*Document method*), 170
- xref_get_keys() (*Document method*), 169
- xref_length() (*Document method*), 370
- xref_object() (*Document method*), 188
- xref_set_key() (*Document method*), 171
- xref_stream() (*Document method*), 188
- xref_stream_raw() (*Document method*), 188
- xref_xml_metadata() (*Document method*), 193
- xrefLength, 451
- xrefObject, 451
- xrefs
 - get_image_info, 250
- xrefStream, 452
- xrefStreamRaw, 452
- xres (*Pixmap attribute*), 277

Y

- y (*Pixmap attribute*), 277
- y (*Point attribute*), 280
- y0 (*IRect attribute*), 212
- y0 (*Rect attribute*), 290
- y1 (*IRect attribute*), 212
- y1 (*Rect attribute*), 290
- yres (*Pixmap attribute*), 277

Z

- zoom, 17
 - resolution, 17