PyMuPDF

PyMuPDF Documentation

Artifex

Sep 03, 2026

ABOUT

1 Features Comparison 3

2 PyMuPDF Product Suite 5

3 Performance 7

4 License and Copyright 9

5 PyMuPDF4LLM 11

6 PyMuPDF Pro 17

7 Installation 21

8 The Basics 27

9 Tutorial 47

10 PyMuPDF, LLM & RAG 57

11 Resources 59

12 FAQ 61

13 Opening Files 63

14 Converting Files 67

15 Compressing Files 71

16 OCR - Optical Character Recognition 77

17 Text 79

18 Images 97

19 Annotations 113

20 Drawing and Graphics 123

21 Stories 131

22 Journalling 175

23 Multiprocessing 179

24 Optional Content Support 185

25 Low-Level Interfaces 187

26 Common Issues and their Solutions 197

27 Classes 203

28 The PyMuPDF4LLM API 445

29 Operator Algebra for Geometry Objects 457

30 Low Level Functions and Classes 463

31 Glossary 487

32 Constants and Enumerations 493

33 Color Database 505

34 Appendix 1: Details on Text Extraction 507

35 Appendix 2: Considerations on Embedded Files 515

36 Appendix 3: Assorted Technical Information 517

37 Appendix 4: Performance Comparison Methodology 529

38 Change Log 535

39 Deprecated Names 595

Index 597

is a high-performance **Python** library for data extraction, analysis, conversion & manipulation of (and other) documents.
is hosted on GitHub and registered on PyPI.

This documentation covers all versions up to 1.28.2.

CHAPTER

ONE

FEATURES COMPARISON

1.1 Feature Matrix

The following table illustrates how compares with other typical solutions.

Note

A note about **Office** document types (DOCX, XLXS, PPTX) and **Hangul** documents (HWPX). These documents can be loaded into and you will receive a *Document* object.

There are some caveats:

- we convert the input to **HTML** to layout the content.
- because of this the original page separation has gone.

When saving out the result any faithful representation of the original layout cannot be expected.

Therefore input files are mostly in a form that's useful for text extraction.

If faithful reproduction of layout is important, then consider using *PyMuPDF Pro*.

CHAPTER

TWO

PYMUPDF PRODUCT SUITE

is the standard version of the library, however there are a family of additional products each with different features and functionality.

Additional products in the product suite are:

- adds support for Office document formats.
- is optimized for large language model (LLM) applications, providing enhanced text extraction and processing capabilities.

  It focuses on layout analysis and semantic understanding, ideal for document conversion and formatting tasks with enhanced results.

> **Note**
>
> All of the products above depend on the same core product - and therefore have full access to all of its features. These additional products can be seen as optional extras to the enhance the core library.

2.1 PyMuPDF Products Comparison

The following table illustrates what features the products offer:

Table 1: PyMuPDF Products Comparison

	PyMuPDF	PyMuPDF Pro	PyMuPDF4LLM
Input Documents	`PDF`, `XPS`, `EPUB`, `CBZ`, `MOBI`, `FB2`, `SVG`, `TXT`, `MD`, Images (*standard document types*)	*as PyMuPDF* and: `DOC/DOCX`, `XLS/XLSX`, `PPT/PPTX`, `HWP/HWPX`	*as PyMuPDF*
Output Documents	Can convert any input document to `PDF`, `SVG` or Image	*as PyMuPDF*	*as PyMuPDF* and: Markdown (`MD`), `JSON` or `TXT`
Page Analysis	Basic page analysis to return document structure	*as PyMuPDF*	Advanced Page Analysis with trained data for enhanced results
Data extraction	Basic data extraction with structured layout information and bounding box data	*as PyMuPDF*	Advanced data extraction including layout analysis with semantic understanding and enhanced bounding box data
Table extraction	Basic table extraction as part of text extraction	*as PyMuPDF*	Advanced table extraction with cell structure, including support for merged cells and complex layouts
Image extraction	Basic image extraction	*as PyMuPDF*	Advanced detection and rendering of image areas on page saving them to disk or embedding in MD output
Vector extraction	Vector extraction and clustering	*as PyMuPDF*	Superior detection of "picture" areas
Popular RAG Integrations	Langchain, LlamaIndex	*as PyMuPDF*	*as PyMuPDF* and with some additional help methods for RAG workflows
OCR	On-demand invocation of built-in Tesseract for text detection on pages or images	*as PyMuPDF*	Automatic OCR based on page content analysis. OCR adapators for popular OCR engines available

CHAPTER

THREE

PERFORMANCE

To benchmark performance against a range of tasks a test suite with a fixed set of *8 PDFs with a total of 7,031 pages* containing text & images is used to obtain performance timings.

Here are current results, grouped by task:

> **Note**
>
> For more detail regarding the methodology for these performance timings see: *Performance Comparison Methodology*.

CHAPTER

FOUR

LICENSE AND COPYRIGHT

and are now available under both, open-source and commercial license agreements. Please read the full text of the license agreement, available in the distribution material (file COPYING) and on the GNU license page, to ensure that your use case complies with the guidelines of the license. If you determine you cannot meet the requirements of the , 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.*

This documentation covers PyMuPDF 1.28.2.

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.

Typically PyMuPDF is released more frequently than MuPDF so it will often be the case that the patch level of PyMuPDF will be greater than the embedded MuPDF.

For example PyMuPDF-1.24.5 contains MuPDF-1.24.2.

Also see `pymupdf_version` and `mupdf_version`.

This documentation covers all versions up to 1.28.2.

CHAPTER

FIVE

PYMUPDF4LLM

is a lightweight extension for that turns PDFs into clean, structured data with minimal setup. It includes layout analysis *without* any GPU requirement.

makes it easy to extract document content in the format you need for **LLM** & **RAG** environments. It supports structured data extraction to *Markdown*, *JSON* and *TXT*, as well as *LlamaIndex* and *LangChain* integration.

Important

You can also extend the supported file types to also include **Office** document formats (DOC/DOCX, XLS/XLSX, PPT/PPTX, HWP/HWPX) by *using PyMuPDF Pro with PyMuPDF4LLM*.

5.1 Features

- Support for Markdown, JSON and plain text output formats.
- Support for multi-column pages.
- Support for image and vector graphics extraction.
- Layout analysis for better semantic understanding of document structure.
- Support for page chunking output.
- Automatic detection of pages which profit from OCR and support for various OCR engines.
- Integration with *LlamaIndex* & *LangChain*.

5.2 API

See: *The PyMuPDF4LLM API*.

5.3 Installation

Install the package via **pip** with:

```
pip install pymupdf4llm
```

5.4 Extracting

5.4.1 As Markdown

To retrieve your document content in **Markdown** use the *to_markdown()* method as follows:

```
import pymupdf4llm
md = pymupdf4llm.to_markdown("input.pdf")
```

5.4.2 As JSON

To retrieve your document content in **JSON** use the *to_json()* method as follows:

```
import pymupdf4llm
json = pymupdf4llm.to_json("input.pdf")
```

The JSON export will give you bounding box information and layout data for each element on the page. This can be used to create your own custom output formats or to simply have more detailed information about the document structure for RAG workflows & LLM integrations.

5.4.3 As TXT

To retrieve your document content in **TXT** use the *to_text()* method as follows:

```
import pymupdf4llm
txt = pymupdf4llm.to_text("input.pdf")
```

Note

Instead of using filename strings as above, one can also provide a *PyMuPDF Document*.

Finally we can save the output to an external file as follows:

```
from pathlib import Path
suffix = ".md" # or ".json" or ".txt"
Path(doc.name).with_suffix(suffix).write_bytes(md.encode())
```

5.4.4 Headers & Footers

Many documents will have header and footer information on each page of a PDF which you may or may not want to include. This information can be repetitive and simply not needed (e.g. the same logo and document title or page number information is not always required when it comes to extracting the document content).

is trained in detecting these typical document elements and able to omit them.

So in this case we can adjust our API calls to ignore these elements as follows:

```
md = pymupdf4llm.to_markdown(doc, header=False, footer=False)
```

> **Note**
>
> Please note that page `header` / `footer` exclusion is not applicable to JSON output as it aims to always represent all data for the included pages. Please refer to *The PyMuPDF4LLM API* for more.

5.5 Integrations

5.5.1 With LlamaIndex

supports direct conversion to a **LlamaIndex** document. A document is first converted into **Markdown** format and then a **LlamaIndex** document is returned as follows:

```
import pymupdf4llm
llama_reader = pymupdf4llm.LlamaMarkdownReader()
llama_docs = llama_reader.load_data("input.pdf")
```

5.5.2 With LangChain

also supports **LangChain** integration, see the PyMuPDF4LLM Document Loader for more details.

5.6 Using with

For **Office** document support, works seamlessly with . Assuming you have *PyMuPDF Pro* installed you will be able to work with **Office** documents as expected:

```
import pymupdf4llm
import pymupdf.pro
pymupdf.pro.unlock()
md = pymupdf4llm.to_markdown("sample.doc")
```

5.7 PyMuPDF4LLM & PyMuPDF Layout

By default includes a layout analysis module to enhance output results. To disable this module you can do so by calling the *`use_layout()`* method.

5.8 OCR

PyMuPDF4LLM includes built-in OCR support for scanned documents and image-based PDFs. By default, OCR runs **automatically** when needed — you don't have to opt in. For more control, you can force OCR on specific pages, disable it entirely, or swap in a different OCR engine using the adaptor interface.

> **Note**
>
> If you want to use an OCR engine other than Tesseract, see *OCR Engines* for details.

5.8.1 Hybrid OCR strategy

PyMuPDF4LLM applies OCR only when it is genuinely required to obtain the complete text of a PDF page. If a page already contains sufficient extractable text, OCR is skipped entirely — avoiding unnecessary work and eliminating the risk of degrading high-quality digital text.

When OCR is needed, PyMuPDF4LLM automatically selects the most suitable OCR plugin available in the runtime environment, balancing detection accuracy with processing speed.

Its built-in OCR plugins implement a Hybrid OCR strategy: only those regions lacking extractable, legible text are passed to the OCR engine. This selective approach typically reduces OCR processing time by around 50% while improving recognition accuracy, since the engine focuses exclusively on the problematic regions. The recognized text is then merged back into the original page, enriching it without disturbing existing digital content.

5.8.2 Auto-OCR Behaviour

PyMuPDF4LLM inspects each page before extracting text. If a page contains **no selectable text** — meaning all content is rasterised into images — OCR is triggered automatically for that page.

Pages that contain native text only are never sent through OCR. This keeps processing fast and avoids degrading already-clean text.

```
import pymupdf4llm

# OCR runs automatically on any page with no selectable text
md_text = pymupdf4llm.to_markdown("scanned-document.pdf")
```

The resulting Markdown is seamless — pages extracted via OCR and pages extracted natively are combined into a single output with no distinction between them.

5.8.3 How OCR is Triggered

There are two scenarios where OCR is applied automatically:

No text at all — if a page contains roughly no text but is covered with images or many character-sized vectors, PyMuPDF4LLM checks whether text is *probably* detectable on the page. This distinguishes image-based text (e.g. a scanned document) from ordinary pictures like photographs.

Garbled text — if a page does contain text but too many characters are unreadable (e.g. ""), OCR is applied **for the affected text areas only**, not the full page. This preserves already-readable text, images, and vectors while recovering only what is broken.

5.8.4 Forcing OCR

In some cases you may want to force OCR even on pages that contain selectable text — for example, when the native text layer is corrupt, misencoded, or misaligned with the visual content.

Use `force_ocr=True` to bypass the auto-detection check entirely:

```
md_text = pymupdf4llm.to_markdown("document.pdf", force_ocr=True)
```

> **Warning**
>
> Forcing OCR on clean, text-based PDFs will slow down processing significantly and may reduce output quality. Only use `force_ocr=True` when you have reason to distrust the native text layer.

You can also force OCR on specific pages rather than the whole document:

```
md_text = pymupdf4llm.to_markdown(
    "document.pdf",
    pages=[2, 3, 4],
    force_ocr=True
)
```

5.8.5 Disabling OCR

To prevent OCR from running at all — even on pages with no selectable text — set `use_ocr=False`:

```
md_text = pymupdf4llm.to_markdown("document.pdf", use_ocr=False)
```

Pages with no selectable text will return empty strings in this mode. This is useful when you know your documents are always text-based, or when you want to handle OCR yourself in a downstream step.

5.8.6 OCR Engines

Other OCR Engines (OCR Adaptors or Plugins) can be used with PyMuPDF4LLM.

See ocr-plugins for details on how to use different OCR engines with PyMuPDF4LLM, including Tesseract, RapidOCR, and how to implement your own custom OCR function.

5.8.7 OCR Language Support

When using the default Tesseract adaptor, you can specify one or more languages using Tesseract's language codes.

Specify the language to be used by the Tesseract OCR engine. Default is `"eng"` (English). Make sure that the respective language data files are installed. Remember to use correct Tesseract language codes. Multiple languages can be specified by concatenating the respective codes with a plus sign `"+"`, for example `"eng+deu"` for English and German.

```
md_text = pymupdf4llm.to_markdown("multilingual.pdf",
                                  ocr_language="eng+deu")
```

Tesseract language packs must be installed on your system. For example, on Ubuntu:

```
sudo apt install tesseract-ocr-deu tesseract-ocr-fra
```

See the page on installing Tesseract language packs for further details.

5.8.8 Performance Tips

OCR is the most compute-intensive part of the extraction pipeline. A few ways to keep it fast:

- **Process only the pages you need** using the `pages` parameter to avoid running OCR on the entire document.
- **Cache results** — write the output to disk after the first run so you don't re-process the same file.
- **Use** `force_ocr=False` (the default) so clean pages skip OCR entirely.
- **Resize images before passing to OCR** — very high DPI scans can slow Tesseract down without improving accuracy.

5.9 Further Resources

5.9.1 Sample code

- Command line RAG Chatbot with PyMuPDF
- Example of a Browser Application using LangChain and PyMuPDF

5.9.2 Blogs

- RAG/LLM and PDF: Enhanced Text Extraction
- Creating a RAG Chatbot with ChatGPT and PyMuPDF
- Building a RAG Chatbot GUI with the ChatGPT API and PyMuPDF
- RAG/LLM and PDF: Conversion to Markdown Text with PyMuPDF

This documentation covers all versions up to 1.28.2.

CHAPTER

SIX

PYMUPDF PRO

is a set of *commercial extensions* for .

Enhance capability with **Office** document support & **RAG/LLM** integrations.

- Enables Office document handling, including `doc`, `docx`, `hwp`, `hwpx`, `ppt`, `pptx`, `xls`, `xlsx`, and others.
- Supports text and table extraction, document conversion and more.
- Includes the commercial version of .

To enquire about obtaining a commercial license, then use this contact page.

Trial license keys are available for evaluation purposes. Please fill out the form on this page to obtain a trial key.

Note

A licensed version of also gives you a licensed version of . If you are interested in using the package you should install it separately.

6.1 Platform support

Available for these platforms only:

- Windows x86_64.
- Linux x86_64 (glibc).
- MacOS x86_64.
- MacOS arm64.

6.2 Office file support

In addition to the *standard file types supported by PyMuPDF*, supports:

DOC/DOCX	XLS/XLSX	PPT/PPTX	HWP/HWPX

6.3 Usage

6.3.1 Installation

Install via pip with:

```
pip install pymupdfpro
```

6.3.2 Loading an Office document

Import and you can then reference **Office** documents directly, e.g.:

```
import pymupdf.pro
pymupdf.pro.unlock()
# PyMuPDF has now been extended with PyMuPDF Pro features, with some restrictions.
doc = pymupdf.open("my-office-doc.xls")
```

> **Note**
>
> All standard functionality is exposed as expected - handles the extended **Office** file types

From then on you can work with document pages just as you would do normally, but with respect to the *restrictions*.

6.3.3 Restrictions

functionality is restricted without a license key as follows:

> **Only the first 3 pages of any document will be available.**
>
> To unlock full functionality you should obtain a trial key.

6.4 Trial keys

To obtain a license key please fill out the form on this page. You will then have the trial key emailled to the address you submitted.

6.4.1 Using a key

Initialize with a key as follows:

```
import pymupdf.pro
pymupdf.pro.unlock(my_key)
# PyMuPDF has now been extended with PyMuPDF Pro features.
```

This will allow you to evaluate the product for a limited time. If you want to use after this time you should then enquire about obtaining a commercial license.

6.5 Converting

6.5.1 Office document to

Use the *office_to_pdf()* method to convert an **Office** document to , e.g.:

```
import pymupdf.pro
pymupdf.pro.unlock()

pymupdf.pro.office_to_pdf("input.docx", "output.pdf")
```

If you require a byte representation of the PDF data, you can use the *office_to_pdf()* without specifying an output file, e.g.:

```
import pymupdf.pro
pymupdf.pro.unlock()

pdfdata = pymupdf.pro.office_to_pdf("input.docx")
```

6.5.2 Office document to Images

In order to convert an **Office** document to images, you should iterate the document pages, convert to *Pixmap* and save, e.g.:

```
doc = pymupdf.open("input.docx")

for i, page in enumerate(doc):
    pix = page.get_pixmap(dpi=200)
    pix.save(f"page-{i+1}.png")

doc.close()
```

6.5.3 Office document to

Use the *office_to_markdown()* method to convert an **Office** document to , e.g.:

```
import pymupdf.pro
pymupdf.pro.unlock()

pymupdf.pro.office_to_markdown("input.docx", "output.md")
```

If you require a string representation of the Markdown data, you can use the *office_to_markdown()* without specifying an output file.

6.5.4 Office document to

Use the *office_to_json()* method to convert an **Office** document to , e.g.:

```
import pymupdf.pro
pymupdf.pro.unlock()

pymupdf.pro.office_to_json("input.docx", "output.json")
```

If you require a string representation of the JSON data, you can use the *office_to_json()* without specifying an output file.

6.6 Fonts

By default `pymupdf.pro.unlock()` searches for all installed font directories.

This can be controlled with keyword-only args:

- `fontpath`: specific font directories, either as a list/tuple or `os.sep`-separated string. If None (the default), we use `os.environ['PYMUPDFPRO_FONT_PATH']` if set.
- `fontpath_auto`: Whether to append system font directories. If None (the default) we use true if `os.environ['PYMUPDFPRO_FONT_PATH_AUTO']` is '1'. If true we append all system font directories.

Function `pymupdf.pro.get_fontpath()` returns a tuple of all font directories used by `unlock()`.

6.7 API

office_to_pdf(*input_path: str*, *output_path: str = None*) → bytes | None

Reads the input file and converts its contents into format.

Parameters

- **input_path** (*str*) – the input file path.
- **output_path** (*str*) – the path to the output file. If None, the method returns the bytes of the PDF content.

Returns

Either bytes of the PDF content, None if output_path is specified.

office_to_markdown(*input_path: str*, *output_path: str = None*) → str | None

Reads the input file and outputs the text of its pages in format.

Parameters

- **input_path** (*str*) – the input file path.
- **output_path** (*str*) – the path to the output file. If None, the method returns the content as a string.

Returns

Either a string of the Markdown representation, None if output_path is specified.

office_to_json(*input_path: str*, *output_path: str = None*) → str | None

Reads the input file and outputs the text of its pages in format.

Parameters

- **input_path** (*str*) – the input file path.
- **output_path** (*str*) – the path to the output file. If None, the method returns the content as a string.

Returns

Either a string of the JSON representation, None if output_path is specified.

This documentation covers all versions up to 1.28.2.

CHAPTER

SEVEN

INSTALLATION

7.1 Requirements

All the examples below assume that you are running inside a Python virtual environment. See: https://docs.python.org/3/library/venv.html for details. We also assume that `pip` is up to date.

For example:

- Windows:

```
py -m venv pymupdf-venv
.\pymupdf-venv\Scripts\activate
python -m pip install --upgrade pip
```

- Linux, MacOS:

```
python -m venv pymupdf-venv
. pymupdf-venv/bin/activate
python -m pip install --upgrade pip
```

7.2 Installation

should be installed using pip with:

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

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

7.3 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 to be installed:

- 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.
  - Having other installed versions of Visual Studio, for example Visual Studio 2022, can cause problems because one can end up with MuPDF and PyMuPDF code being compiled with different compiler versions.

The build will automatically download and build MuPDF.

7.4 Problems after installation

- On Windows, Python error:

```
ImportError: DLL load failed while importing _extra
```

  This has been occasionally seen if `MSVCP140.dll` is missing, and appears to be caused by a bug in some versions (2015-2017) of `Microsoft Visual C++ Redistributables`.

  It is recommended to search for `MSVCP140.dll` in https://msdn.com to find instructions for how to reinstall it. For example https://learn.microsoft.com/cpp/windows/latest-supported-vc-redist has permalinks to the latest supported versions.

  See https://github.com/pymupdf/PyMuPDF/issues/2678 for more details.

- Python error:

```
ModuleNotFoundError: No module named 'frontend'
```

  This can happen if PyMuPDF's legacy name `fitz` is used (for example `import fitz` instead of `import pymupdf`), and an unrelated Python package called `fitz` (https://pypi.org/project/fitz/) is installed.

  The fitz package appears to be no longer maintained (the latest release is from 2017), but unfortunately it does not seem possible to remove it from pypi.org. It does not even work on its own, as well as breaking the use of PyMuPDF's legacy name.

  There are a few ways to avoid this problem:

  - Use `import pymupdf` instead of `import fitz`, and update one's code to match.
  - Or uninstall the `fitz` package and reinstall PyMuPDF:

```
pip uninstall fitz
pip install --force-reinstall pymupdf
```

  - Or use `import pymupdf as fitz`. However this has not been well tested.

- With Jupyter labs on Apple Silicon (arm64), Python error:

```
ImportError: /opt/conda/lib/python3.11/site-packages/pymupdf/libmupdf.so.24.4:␣
↪undefined symbol: fz_pclm_write_options_usage
```

  This appears to be a problem in Jupyter labs; see: https://github.com/pymupdf/PyMuPDF/issues/3643#issuecomment-2210588778.

- On Windows, Python error:

```
ImportError: dynamic module does not define module export function (PyInit__extra)
```

  This was reported 2025-03-26 in https://github.com/pymupdf/PyMuPDF/issues/4405.

  The fix appears to be to install the latest `VC_redist.x64.exe`.

7.5 Notes

- Wheels are available for the following platforms:
  - Windows 32-bit Intel.
  - Windows 64-bit Intel.

  - Linux 64-bit Intel.
  - Linux 64-bit ARM.
  - MacOS 64-bit Intel.
  - MacOS 64-bit ARM.

  Details:

  - We release a single wheel for each of the above platforms.
  - Each wheel uses the Python Stable ABI of the current oldest supported Python version (currently 3.10), and so works with all later Python versions, including new Python releases.
  - Wheels are tested on all Python versions currently marked as "Supported" on https://devguide.python.org/versions/, currently 3.9, 3.10, 3.11, 3.12, 3.13 and 3.14.

- 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
- Wheels are not available for Linux-aarch64 with Musl libc (For example Alpine Linux on aarch64), and building from source is known to fail.
- 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, Tesseract must be installed and the `tessdata` folder name specified; 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.

7.6 Build and install from a local PyMuPDF source tree

Initial setup:

- Install C/C++ development tools as described above.
- Enter a Python venv and update pip, as described above.
- Get a PyMuPDF source tree:
  - Clone the PyMuPDF git repository:

    ```
    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.

Then one can build PyMuPDF in two ways:

- Build and install PyMuPDF with default MuPDF version:

```
cd PyMuPDF && pip install .
```

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

- Or build and install PyMuPDF using a local MuPDF source tree:
  - Clone the MuPDF git repository:

    ```
    git clone --recursive https://git.ghostscript.com/mupdf.git
    ```

  - Build PyMuPDF, specifying the location of the local MuPDF tree with the environmental variables `PYMUPDF_SETUP_MUPDF_BUILD`:

    ```
    cd PyMuPDF && PYMUPDF_SETUP_MUPDF_BUILD=../mupdf pip install .
    ```

Also, one can build for different Python versions in the same PyMuPDF tree:

- PyMuPDF will build for the version of Python that is being used to run `pip`. To run `pip` with a specific Python version, use `python -m pip` instead of `pip`.

  So for example on Windows one can build different versions with:

  ```
  cd PyMuPDF && py -3.10 -m pip install .
  ```

  or:

  ```
  cd PyMuPDF && py -3.10-32 -m pip install .
  ```

7.7 Running tests

Having a PyMuPDF tree available allows one to run PyMuPDF's `pytest` test suite:

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

7.7.1 Notes about using a non-default MuPDF

Using a non-default build of MuPDF by setting environmental variable `PYMUPDF_SETUP_MUPDF_BUILD` can cause various things to go wrong and so is not generally supported:

- If MuPDF's major version number differs from what PyMuPDF uses by default, PyMuPDF can fail to build, because MuPDF's API can change between major versions.
- Runtime behaviour of PyMuPDF can change because MuPDF's runtime behaviour changes between different minor releases. This can also break some PyMuPDF tests.
- If MuPDF was built with its default config instead of PyMuPDF's customised config (for example if MuPDF is a system install), it is possible that `tests/test_textbox.py:test_textbox3()` will fail. One can skip this particular test by adding `-k 'not test_textbox3'` to the `pytest` command line.

7.8 Official PyMuPDF Linux wheels may not install on older Linux systems

Releases of PyMuPDF are incompatible with older Linux systems.

For example as of 2025-09-03, `pip install pymupdf` does not work on some AWS Lambda systems - see https://github.com/pymupdf/PyMuPDF/discussions/4631.

This is because official PyMuPDF Linux wheels are built with a version of glibc determined by the current Python manylinux environment. These wheels are incompatible with Linux systems that have an older glibc.

The official Python manylinux environment is updated periodically to use newer glibc versions, so new releases of PyMuPDF become increasingly incompatible with older Linux systems.

There is nothing that can be done about this, other than updating older Linux systems, or building PyMuPDF locally from source.

For more details, please see: Python Packaging Authority.

7.9 Packaging

See packaging.

7.10 Using with Pyodide

See pyodide.

7.11 Enabling Integrated OCR Support

PyMuPDF will already contain all the logic to support OCR functions. But it additionally does need Tesseract's language support data.

If not specified explicitly, PyMuPDF will attempt to find the installed Tesseract's `tessdata`, but this should probably not be relied upon.

Otherwise PyMuPDF requires that Tesseract's language support folder is specified explicitly either in PyMuPDF OCR functions' `tessdata` arguments or `os.environ["TESSDATA_PREFIX"]`.

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

1. 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`
2. Specify the language support folder when calling PyMuPDF OCR functions:
    - Set the `tessdata` argument.
    - Or set `os.environ["TESSDATA_PREFIX"]` from within Python.
    - Or set environment variable `TESSDATA_PREFIX` before running Python, for example:
        - Windows: `setx TESSDATA_PREFIX "C:/Program Files/Tesseract-OCR/tessdata"`
        - Unix systems: `declare -x TESSDATA_PREFIX=/usr/share/tesseract-ocr/4.00/tessdata`

> **Note**
>
> English language support is included by default in Tesseract installation.
>
> Tesseract Language Packs for other languages must be installed separately, and the `tessdata` folder must be specified to PyMuPDF as described above, for OCR to work with those languages.

> **Note**
>
> Find out more on the official documentation for installing Tesseract website.

This documentation covers all versions up to 1.28.2.

CHAPTER

EIGHT

THE BASICS

8.1 Opening a File

To open a file, do the following:

```
import pymupdf

doc = pymupdf.open("a.pdf") # open a document
```

Note

Taking it further

See the *list of supported file types* and *The How to Guide on Opening Files* for more advanced options.

8.2 Extract text from a

To extract all the text from a file, do the following:

```
import pymupdf

doc = pymupdf.open("a.pdf") # open a document
out = open("output.txt", "wb") # create a 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()
doc.close()
```

Of course it is not just which can have text extracted - all the *supported document file formats* such as *MOBI*, *EPUB*, *TXT* can have their text extracted.

Note

Taking it further

If your document contains image based text content the use OCR on the page for subsequent text extraction:

```
tp = page.get_textpage_ocr()
text = page.get_text(textpage=tp)
```

There are many more examples which explain how to extract text from specific areas or how to extract tables from documents. Please refer to the *How to Guide for Text*.

You can now also *extract text in Markdown format*.

API reference

- *Page.get_text()*

8.3 Extract images from a

To extract all the images from a file, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open a document

for page_index in range(len(doc)): # iterate over pdf pages
    page = doc[page_index] # get the page
    image_list = page.get_images()

    # print the number of images found on the page
    if image_list:
        print(f"Found {len(image_list)} images on page {page_index}")
    else:
        print("No images found on page", page_index)

    for image_index, img in enumerate(image_list, start=1): # enumerate the image list
        xref = img[0] # get the XREF of the image
        pix = pymupdf.Pixmap(doc, xref) # create a Pixmap

        if pix.n - pix.alpha > 3: # CMYK: convert to RGB first
            pix = pymupdf.Pixmap(pymupdf.csRGB, pix)

        pix.save(f"page_{page_index}-image_{image_index}.png") # save the image as png
        pix = None
```

Note

Taking it further

There are many more examples which explain how to extract text from specific areas or how to extract tables from documents. Please refer to the *How to Guide for Text*.

API reference

- *Page.get_images()*
- *Pixmap*

8.4 Extract vector graphics

To extract all the vector graphics from a document page, do the following:

```
doc = pymupdf.open("some.file")
page = doc[0]
paths = page.get_drawings()
```

This will return a dictionary of paths for any vector drawings found on the page.

Note

Taking it further

Please refer to: *How to Extract Drawings*.

API reference

- *Page.get_drawings()*

8.5 Merging files

To merge files, do the following:

```
import pymupdf

doc_a = pymupdf.open("a.pdf") # open the 1st document
doc_b = pymupdf.open("b.pdf") # open the 2nd document

doc_a.insert_pdf(doc_b) # merge the docs
doc_a.save("a+b.pdf") # save the merged document with a new filename
```

8.5.1 Merging files with other types of file

With *Document.insert_file()* you can invoke the method to merge *supported files* with . For example:

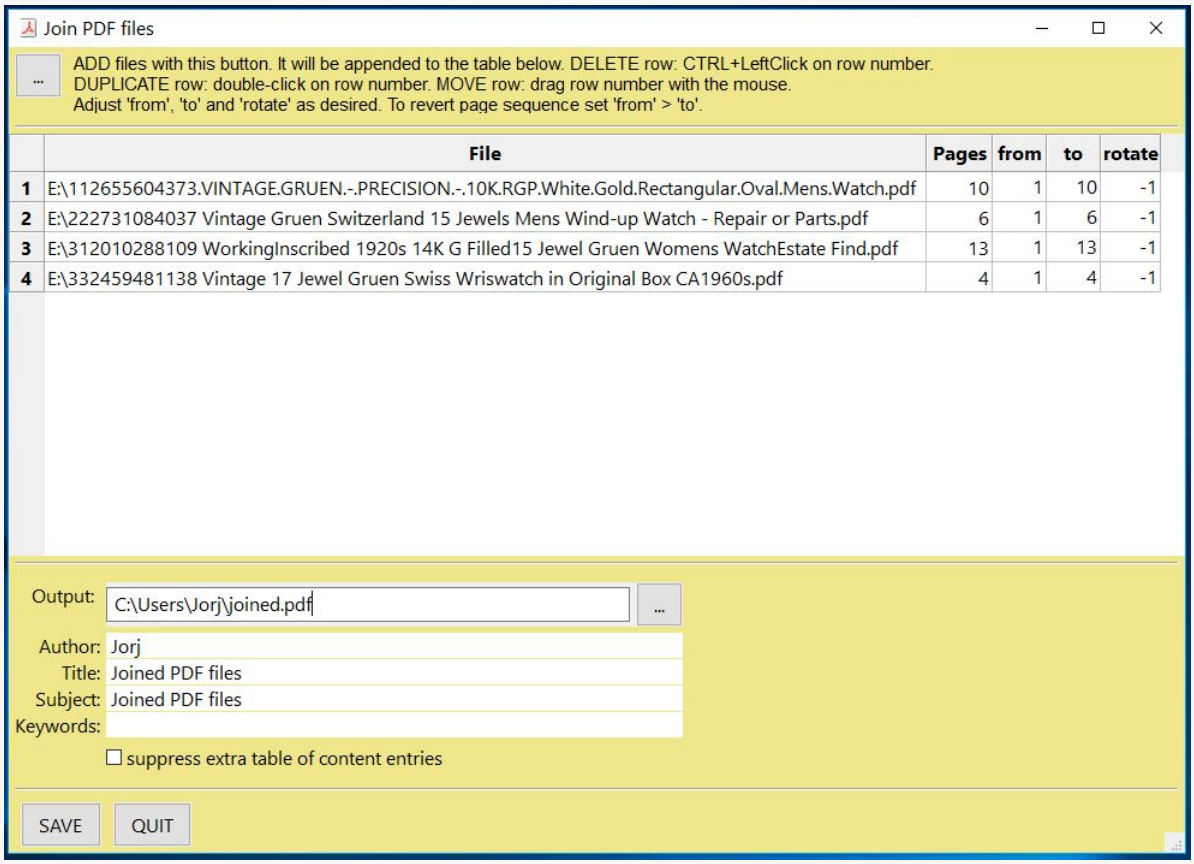
```
import pymupdf

doc_a = pymupdf.open("a.pdf") # open the 1st document
doc_b = pymupdf.open("b.svg") # open the 2nd document

doc_a.insert_file(doc_b) # merge the docs
doc_a.save("a+b.pdf") # save the merged document with a new filename
```

Note

Taking it further

It is easy to join PDFs with *Document.insert_pdf()* & *Document.insert_file()*. Given open 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.

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



API reference

- *Document.insert_pdf()*
- *Document.insert_file()*

8.6 Working with Coordinates

There is one *mathematical term* that you should feel comfortable with when using - **"coordinates"**. Please have a quick look at the *Coordinates* section to understand the coordinate system to help you with positioning objects and understand your document space.

8.7 Adding a watermark to a

To add a watermark to a file, do the following:

```
import pymupdf
```

(continues on next page)

(continued from previous page)

```
doc = pymupdf.open("document.pdf") # open a document

for page_index in range(len(doc)): # iterate over pdf pages
    page = doc[page_index] # get the page

    # insert an image watermark from a file name to fit the page bounds
    page.insert_image(page.bound(),filename="watermark.png", overlay=False)

doc.save("watermarked-document.pdf") # save the document with a new filename
```

Note

Taking it further

Adding watermarks is essentially as simple as adding an image at the base of each page. You should ensure that the image has the required opacity and aspect ratio to make it look the way you need it to.

In the example above a new image is created from each file reference, but to be more performant (by saving memory and file size) this image data should be referenced only once - see the code example and explanation on *Page.insert_image()* for the implementation.

API reference

- *Page.bound()*
- *Page.insert_image()*

8.8 Adding an image to a

To add an image to a file, for example a logo, do the following:

```
import pymupdf

doc = pymupdf.open("document.pdf") # open a document

for page_index in range(len(doc)): # iterate over pdf pages
    page = doc[page_index] # get the page

    # insert an image logo from a file name at the top left of the document
    page.insert_image(pymupdf.Rect(0,0,50,50),filename="my-logo.png")

doc.save("logo-document.pdf") # save the document with a new filename
```

Note

Taking it further

As with the watermark example you should ensure to be more performant by only referencing the image once if possible - see the code example and explanation on *Page.insert_image()*.

API reference

- *Rect*
- *Page.insert_image()*

8.9 Rotating a

To add a rotation to a page, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open document
page = doc[0] # get the 1st page of the document
page.set_rotation(90) # rotate the page
doc.save("rotated-page-1.pdf")
```

Note

API reference

- *Page.set_rotation()*

8.10 Cropping a

To crop a page to a defined *Rect*, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open document
page = doc[0] # get the 1st page of the document
page.set_cropbox(pymupdf.Rect(100, 100, 400, 400)) # set a cropbox for the page
doc.save("cropped-page-1.pdf")
```

Note

API reference

- *Page.set_cropbox()*

8.11 Attaching Files

To attach another file to a page, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open main document
attachment = pymupdf.open("my-attachment.pdf") # open document you want to attach

page = doc[0] # get the 1st page of the document
point = pymupdf.Point(100, 100) # create the point where you want to add the attachment
attachment_data = attachment.tobytes() # get the document byte data as a buffer

# add the file annotation with the point, data and the file name
file_annotation = page.add_file_annot(point, attachment_data, "attachment.pdf")

doc.save("document-with-attachment.pdf") # save the document
```

Note

Taking it further

When adding the file with *Page.add_file_annot()* note that the third parameter for the `filename` should include the actual file extension. Without this the attachment possibly will not be able to be recognized as being something which can be opened. For example, if the `filename` is just *"attachment"* when view the resulting PDF and attempting to open the attachment you may well get an error. However, with *"attachment.pdf"* this can be recognized and opened by PDF viewers as a valid file type.

The default icon for the attachment is by default a "push pin", however you can change this by setting the `icon` parameter.

API reference

- *Point*
- *Document.tobytes()*
- *Page.add_file_annot()*

8.12 Embedding Files

To embed a file to a document, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open main document
embedded_doc = pymupdf.open("my-embed.pdf") # open document you want to embed

embedded_data = embedded_doc.tobytes() # get the document byte data as a buffer

# embed with the file name and the data
doc.embfile_add("my-embedded_file.pdf", embedded_data)

doc.save("document-with-embed.pdf") # save the document
```

Note

Taking it further

As with *attaching files*, when adding the file with *Document.embfile_add()* note that the first parameter for the `filename` should include the actual file extension.

API reference

- *Document.tobytes()*
- *Document.embfile_add()*

8.13 Deleting Pages

To delete a page from a document, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open a document
doc.delete_page(0) # delete the 1st page of the document
doc.save("test-deleted-page-one.pdf") # save the document
```

To delete a multiple pages from a document, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open a document
doc.delete_pages(from_page=9, to_page=14) # delete a page range from the document
doc.save("test-deleted-pages.pdf") # save the document
```

8.13.1 What happens if I delete a page referred to by bookmarks or hyperlinks?

- A bookmark (entry in the Table of Contents) will become inactive and will no longer navigate to any page.
- A hyperlink will be removed from the page that contains it. The visible content on that page will not otherwise be changed in any way.

Note

Taking it further

The page index is zero-based, so to delete page 10 of a document you would do the following `doc.delete_page(9)`.

Similarly, `doc.delete_pages(from_page=9, to_page=14)` will delete pages 10 - 15 inclusive.

API reference

- *Document.delete_page()*
- *Document.delete_pages()*

8.14 Re-Arranging Pages

To change the sequence of pages, i.e. re-arrange pages, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open a document
doc.move_page(1,0) # move the 2nd page of the document to the start of the document
doc.save("test-page-moved.pdf") # save the document
```

Note

API reference

- *Document.move_page()*

8.15 Copying Pages

To copy pages, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open a document
doc.copy_page(0) # copy the 1st page and puts it at the end of the document
doc.save("test-page-copied.pdf") # save the document
```

Note

API reference

- *Document.copy_page()*

8.16 Selecting Pages

To select pages, do the following:

```
import pymupdf

doc = pymupdf.open("test.pdf") # open a document
doc.select([0, 1]) # select the 1st & 2nd page of the document
doc.save("just-page-one-and-two.pdf") # save the document
```

Note

Taking it further

With you have all options to copy, move, delete or re-arrange the pages of a . 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 = pymupdf.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, ..., n, 0, 1, ..., 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
```

API reference

- *Document.select()*

8.17 Adding Blank Pages

To add a blank page, do the following:

```
import pymupdf

doc = pymupdf.open(...) # some new or existing PDF document
page = doc.new_page(-1, # insertion point: end of document
                    width = 595, # page dimension: A4 portrait
                    height = 842)
doc.save("doc-with-new-blank-page.pdf") # save the document
```

Note

Taking it further

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

```
w, h = pymupdf.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.

Here is a 3-liner that creates a : with one empty page. Its file size is 460 bytes:

```
doc = pymupdf.open()
doc.new_page()
doc.save("A4.pdf")
```

API reference

- *Document.new_page()*
- `paperSizes`

8.18 Inserting Pages with Text Content

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

```
import pymupdf

doc = pymupdf.open(...)  # some new or existing PDF document
n = doc.insert_page(-1, # default insertion point
                    text = "The quick brown fox jumped over the lazy dog",
                    fontsize = 11,
                    width = 595,
                    height = 842,
                    fontname = "Helvetica", # default font
                    fontfile = None, # any font file name
                    color = (0, 0, 0)) # text color (RGB)
```

Note

Taking it further

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.

API reference

- *Document.insert_page()*

8.19 Splitting Single Pages

This deals with splitting up pages of a in arbitrary pieces. For example, you may have a 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 going to a separate page in *Letter* format again.

```
import pymupdf

src = pymupdf.open("test.pdf")
doc = pymupdf.open()  # empty output PDF

for spage in src:  # for each page in input
    r = spage.rect  # input page rectangle
    d = pymupdf.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 = pymupdf.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,
                            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
)
```

Example:

Note

API reference

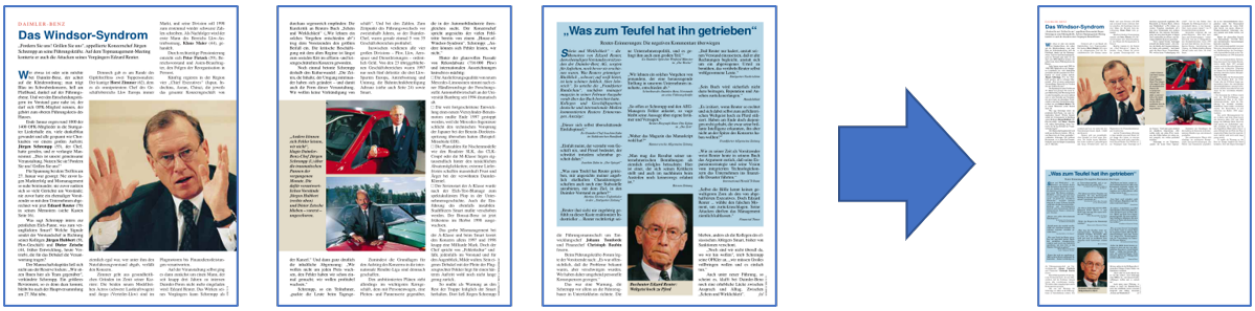
- *Page.cropbox_position()*
- *Page.show_pdf_page()*

8.20 Combining Single Pages

This deals with joining pages to form a new 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.

```
import pymupdf

src = pymupdf.open("test.pdf")
doc = pymupdf.open()  # empty output PDF

width, height = pymupdf.paper_size("a4")  # A4 portrait output page format
r = pymupdf.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 = pymupdf.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.pdf", garbage=3, deflate=True)
```

Example:



Note

API reference

- *Page.cropbox_position()*
- *Page.show_pdf_page()*

8.21 Encryption & Decryption

Starting with version 1.16.0, 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 in a viewer will require providing it.

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

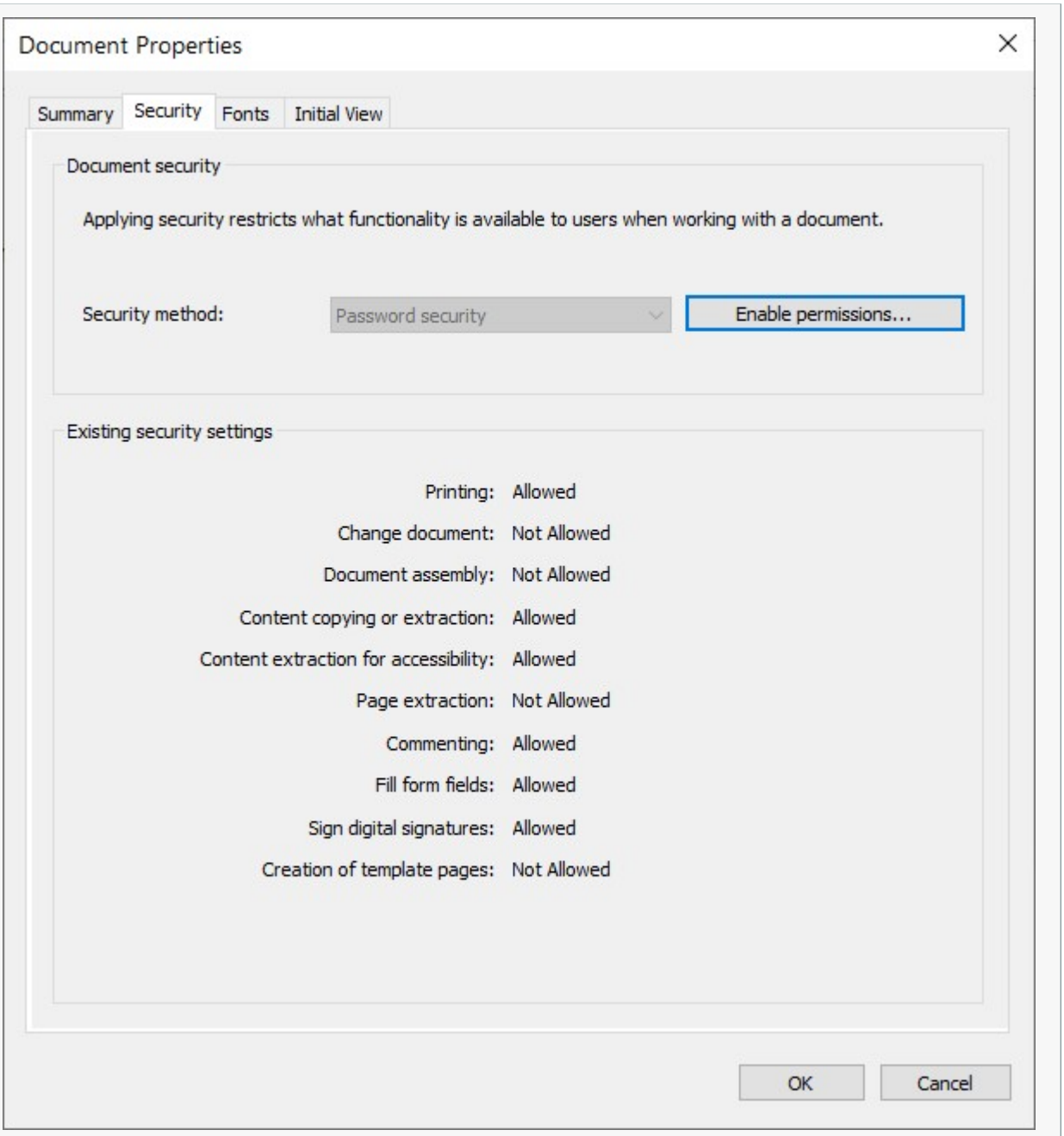
The following snippet creates a new and encrypts it with separate user and owner passwords. Permissions are granted to print, copy and annotate, but no changes are allowed to someone authenticating with the user password.

```
import pymupdf

text = "some secret information" # keep this data secret
perm = int(
    pymupdf.PDF_PERM_ACCESSIBILITY # always use this
    | pymupdf.PDF_PERM_PRINT # permit printing
    | pymupdf.PDF_PERM_COPY # permit copying
    | pymupdf.PDF_PERM_ANNOTATE # permit annotations
)
owner_pass = "owner" # owner password
user_pass = "user" # user password
encrypt_meth = pymupdf.PDF_ENCRYPT_AES_256 # strongest algorithm
doc = pymupdf.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
)
```

Note

Taking it further

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=pymupdf.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.

API reference

- *Document.save()*

8.22 Extracting Tables from a *Page*

Tables can be found and extracted from any document *Page*.

```
import pymupdf
from pprint import pprint

doc = pymupdf.open("test.pdf") # open document
page = doc[0] # get the 1st page of the document
tabs = page.find_tables() # locate and extract any tables on page
print(f"{len(tabs.tables)} found on {page}") # display number of found tables

if tabs.tables:  # at least one table found?
   pprint(tabs[0].extract())  # print content of first table
```

Note

API reference

- *Page.find_tables()*

Important

There is also the pdf2docx extract tables method which is capable of table extraction if you prefer.

8.23 Getting Page Links

Links can be extracted from a *Page* to return *Link* objects.

```
import pymupdf

for page in doc: # iterate the document pages
    link = page.first_link  # a `Link` object or `None`

    while link: # iterate over the links on page
        # do something with the link, then:
        link = link.next # get next link, last one has `None` in its `next`
```

Note

API reference

- *Page.first_link()*

8.24 Getting All Annotations from a Document

Annotations (*Annot*) on pages can be retrieved with the `page.annots()` method.

```
import pymupdf

for page in doc:
    for annot in page.annots():
        print(f'Annotation on page: {page.number} with type: {annot.type} and rect:
↪{annot.rect}')
```

Note

API reference

- *Page.annots()*

8.25 Redacting content from a

Redactions are special types of annotations which can be marked onto a document page to denote an area on the page which should be securely removed. After marking an area with a rectangle then this area will be marked for *redaction*, once the redaction is *applied* then the content is securely removed.

For example if we wanted to redact all instances of the name "Jane Doe" from a document we could do the following:

```
import pymupdf

# Open the PDF document
doc = pymupdf.open('test.pdf')

# Iterate over each page of the document
for page in doc:
    # Find all instances of "Jane Doe" on the current page
    instances = page.search_for("Jane Doe")

    # Redact each instance of "Jane Doe" on the current page
    for inst in instances:
        page.add_redact_annot(inst)

    # Apply the redactions to the current page
    page.apply_redactions()

# Save the modified document
doc.save('redacted_document.pdf')

# Close the document
doc.close()
```

Another example could be redacting an area of a page, but not to redact any line art (i.e. vector graphics) within the defined area, by setting a parameter flag as follows:

```
import pymupdf

# Open the PDF document
doc = pymupdf.open('test.pdf')

# Get the first page
page = doc[0]

# Add an area to redact
rect = [0,0,200,200]

# Add a redacction annotation which will have a red fill color
page.add_redact_annot(rect, fill=(1,0,0))

# Apply the redactions to the current page, but ignore vector graphics
page.apply_redactions(graphics=0)

# Save the modified document
doc.save('redactied_document.pdf')

# Close the document
doc.close()
```

Warning

Once a redacted version of a document is saved then the redacted content in the is *irretrievable*. Thus, a redacted area in a document removes text and graphics completely from that area.

Note

Taking it further

The are a few options for creating and applying redactions to a page, for the full API details to understand the parameters to control these options refer to the API reference.

API reference

- *Page.add_redact_annot()*
- *Page.apply_redactions()*

8.26 Converting PDF Documents

See *Converting Files* for how to convert documents to other formats and vice versa.

8.27 Detecting if a page is black and white

A simple check to determine if a page is pure black and white ("monochrome"). Non-monochrome means that the page has multiple gray levels or colors.

This can be done by using the *Page.get_pixmap()* method to get a pixmap of the page and then checking the *is_monochrome* attribute of the pixmap. We need to ensure that there is no anti-aliasing applied to the pixmap, so we set the anti-aliasing level to 0 using *Tools.set_aa_level()*.

The following example demonstrates how to do this for each page in a document:

```
import pymupdf

pymupdf.TOOLS.set_aa_level(0) # prevent anti-aliasing

def analyze_pdf(path):

    results = []

    with pymupdf.open(path) as doc:
        for i, page in enumerate(doc):
            pix=page.get_pixmap(colorspace=pymupdf.csGRAY)
            bw = pix.is_monochrome

            results.append(bw)
            label = "black & white" if bw==1 else "has color"
            print(f"Page {i + 1}/{len(doc)}: {label}")

    return results

if __name__ == "__main__":
    import sys

    if len(sys.argv) < 2:
        print("Usage: python detect_bw_pages.py <file.pdf>")
        sys.exit(1)

    pdf_path = sys.argv[1]
    analyze_pdf(pdf_path)
```

This documentation covers all versions up to 1.28.2.

CHAPTER

NINE

TUTORIAL

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

Because supports not only PDF, but also XPS, OpenXPS, CBZ, CBR, FB2 and EPUB formats, so does[1]. 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.

In addition to this introduction, please do visit PyMuPDF's YouTube Channel which covers most of the following in the form of YouTube "Shorts" and longer videos.

9.1 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 pymupdf
>>> print(pymupdf.__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).
```

9.1.1 Note on the Name *fitz*

Old versions of had their **Python** import name as `fitz`. Newer versions use `pymupdf` instead, and offer `fitz` as a fallback so that old code will still work.

The reason for the name `fitz` is a historical curiosity:

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).

Note

Use of legacy name `fitz` can fail if defunct pypi.org package `fitz` is installed; see *Problems after installation*.

[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.

9.2 Opening a Document

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

```
doc = pymupdf.open(filename)  # or pymupdf.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.

9.3 Some Document Methods and Attributes

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

9.4 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.

9.5 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.

9.6 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 $-\infty$ < `pno` < `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'
```

(continues on next page)

(continued from previous 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:

9.6.1 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'
```

9.6.2 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.

9.6.3 Saving the Page Image in a File

We can simply store the image in a PNG file:

```
pix.save(f"page-{page.number}.png")
```

9.6.4 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.

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)
```

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 = pymupdf.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.

PyQt4, PyQt5, PySide

Please also see section 3.16 of the Pillow documentation:

```
from PIL import Image, ImageQt

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

Again, you also can get along **without using 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
qtimg = QImage(pix.samples_ptr, pix.width, pix.height, fmt)
```

9.6.5 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 formats[2]:

- **"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 1: Details on Text Extraction*.

9.6.6 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()* .

9.7 Stories: Generating PDF from HTML Source

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, either a Python string or **created by the script** using methods of *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.

It is **not required** to provide syntactically complete HTML documents: snippets like `<b>Hello` are fully accepted, and many / most syntax errors are automatically corrected.

After the HTML is considered complete, it can be used to create a PDF document. This happens via the new *DocumentWriter* class. The programmer calls its methods to create a new empty page, and passes rectangles to the Story to fill them.

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.

9.8 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.

9.8.1 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 sequence[3] of the page numbers that you want to keep. These integers must all be in range *0 <= i < 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).

9.8.2 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 opened 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 = pymupdf.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")
```

[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.

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

9.8.3 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 2: Considerations on Embedded Files*, 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.

9.8.4 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 *xref* 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 -ggggz file.pdf` yields excellent compression results. It corresponds to `doc.save(filename, garbage=4, deflate=True)`.

9.9 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.

9.10 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.

This documentation covers all versions up to 1.28.2.

CHAPTER

TEN

PYMUPDF, LLM & RAG

Integrating into your *Large Language Model (LLM)* framework and overall *RAG (Retrieval-Augmented Generation)* solution provides the fastest and most reliable way to deliver document data.

There are a few well known *LLM* solutions which have their own interfaces with - it is a fast growing area, so please let us know if you discover any more!

If you need to export to *Markdown* or obtain a *LlamaIndex* Document from a file:

10.1 Integration with *LangChain*

It is simple to integrate directly with *LangChain* by using their dedicated loader as follows:

```
from langchain_community.document_loaders import PyMuPDFLoader
loader = PyMuPDFLoader("example.pdf")
data = loader.load()
```

See LangChain Using PyMuPDF for full details.

10.2 Integration with *LlamaIndex*

Use the dedicated `PyMuPDFReader` from *LlamaIndex* to manage your document loading.

```
from llama_index.readers.file import PyMuPDFReader
loader = PyMuPDFReader()
documents = loader.load(file_path="example.pdf")
```

See Building RAG from Scratch for more.

10.3 Preparing Data for Chunking

Chunking (or splitting) data is essential to give context to your *LLM* data and with *Markdown* output now supported by this means that Level 3 chunking is supported.

10.3.1 Outputting as *Markdown*

In order to export your document in *Markdown* format you will need a separate helper. Package *PyMuPDF4LLM* is a high-level wrapper of functions which for each page outputs standard and table text in an integrated Markdown-formatted string across all document pages:

```
# convert the document to markdown
import pymupdf4llm
md_text = pymupdf4llm.to_markdown("input.pdf")

# Write the text to some file in UTF8-encoding
import pathlib
pathlib.Path("output.md").write_bytes(md_text.encode())
```

For further information please refer to: *PyMuPDF4LLM*.

10.3.2 How to use *Markdown* output

Once you have your data in *Markdown* format you are ready to chunk/split it and supply it to your *LLM*, for example, if this is *LangChain* then do the following:

```
import pymupdf4llm
from langchain.text_splitter import MarkdownTextSplitter

# Get the MD text
md_text = pymupdf4llm.to_markdown("input.pdf")  # get markdown for all pages

splitter = MarkdownTextSplitter(chunk_size=40, chunk_overlap=0)

splitter.create_documents([md_text])
```

For more see 5 Levels of Text Splitting

10.4 Related Blogs

To find out more about , *LLM* & *RAG* check out our blogs for implementations & tutorials.

10.4.1 Methodologies to Extract Text

- Enhanced Text Extraction
- Conversion to Markdown Text with PyMuPDF

10.4.2 Create a Chatbot to discuss your documents

- Make a simple command line Chatbot
- Make a Chatbot GUI

This documentation covers all versions up to 1.28.2.

CHAPTER

ELEVEN

RESOURCES

11.1 PyMuPDF Pro

For **Office** file support *try PyMuPDF Pro*.

11.2 Find out about PyMuPDF Utilities

The *GitHub* repository PyMuPDF-Utilities contains a full range of examples, demonstrations and use cases.

11.3 Do you need to DOCX conversion?

We recommend the pdf2docx library which uses and the **python-docx** library to provide simple document conversion from to **DOCX** format.

This documentation covers all versions up to 1.28.2.

CHAPTER

TWELVE

FAQ

CHAPTER

THIRTEEN

OPENING FILES

13.1 Supported File Types

13.1.1 PyMuPDF

can open files other than just .

The following file types are supported:

13.1.2 PyMuPDF Pro

can open Office files.

The following file types are supported:

DOC/DOCX	XLS/XLSX	PPT/PPTX	HWP/HWPX

13.2 How to Open a File

To open a file, do the following:

```
doc = pymupdf.open("a.pdf")
```

> **Note**
>
> The above creates a *Document*. The instruction `doc = pymupdf.Document("a.pdf")` does exactly the same. So, *open()* is just a convenient alias.

To open an empty document, just do:

```
doc = pymupdf.open()
```

13.2.1 File Recognizer: Opening with a Wrong File Extension

If you have a document with a wrong file extension for its type, do not worry: it will still be opened correctly, thanks to the integrated file "content recognizer".

This component looks at the actual data in the file using a number of heuristics – independent of the file extension. This of course is also true for file names **without** an extension.

Here is a list of details about how the file content recognizer works:

- When opening from a file name, use the `filetype` parameter if your file format cannot be determined by content inspection. This is for instance the case for all text files: "txt", "html", "xml" or source files. If the file extension is missing or wrong or the file resides in memory, the `filetype` must be used. File formats that can successfully be recognized will be opened even without or wrong extensions, and the `filetype` paraneter will be ignored.
- Files based on text content do not contain unambiguously recognizable internal structures. This is true for source files (Python, C, etc.) but also HTML, XML and so on. Here, the file extensions and the `filetype` parameter continue to play a role and are used to create a "Tex" / "HTML" / ... document. Correspondingly, text files with other / no extensions, can successfully be opened using `filetype`.

13.3 Opening Remote Files

For remote files on a server (i.e. non-local files), you will need to *stream* the file data to .

For example use the requests library as follows:

```
import pymupdf
import requests

r = requests.get('https://mupdf.com/docs/mupdf_explored.pdf')
data = r.content
doc = pymupdf.Document(stream=data)
```

13.3.1 Opening Files from Cloud Services

For further examples which deal with files held on typical cloud services please see these Cloud Interactions code snippets.

13.4 Opening Django Files

Django implements a File Storage API to store files. The default is the FileSystemStorage, but the django-storages library provides a number of other storage backends.

You can open the file, move the contents into memory, then pass the contents to as a stream.

```
import pymupdf
from django.core.files.storage import default_storage

from .models import MyModel

obj = MyModel.objects.get(id=1)
with default_storage.open(obj.file.name) as f:
```

(continues on next page)

(continued from previous page)

```
    data = f.read()

doc = pymupdf.Document(stream=data)
```

Please note that if the file you open is large, you may run out of memory.

The File Storage API works well if you're using different storage backends in different environments. If you're only using the `FileSystemStorage`, you can simply use the `obj.file.name` to open the file directly with as shown in an earlier example.

13.5 Opening Files as Text

has the capability to open any plain text file as a document. In order to do this you should provide the `filetype` parameter for the `pymupdf.open` function as `"txt"`.

```
doc = pymupdf.open("my_program.py", filetype="txt")
```

In this way you are able to open a variety of file types and perform the typical **non-PDF** specific features like text searching, text extracting and page rendering. Obviously, once you have rendered your `txt` content, then saving as or merging with other files is no problem.

13.5.1 Examples

Opening a C# file

```
doc = pymupdf.open("MyClass.cs", filetype="txt")
```

Opening an XML file

```
doc = pymupdf.open("my_data.xml", filetype="txt")
```

Opening a JSON file

```
doc = pymupdf.open("more_of_my_data.json", filetype="txt")
```

And so on!

As you can imagine many text based file formats can be *very simply opened* and *interpreted* by . This can make data analysis and extraction for a wide range of previously unavailable files possible.

13.6 Full Options for Opening a File

The `pymupdf.open` function has a number of parameters to give you full control over how files are opened. For the full API, please see the *Document* chapter, as *open* is just an alias for the *Document()* constructor.

open(*filename=None*, *stream=None*, *filetype=None*, *archive=None*, *rect=None*, *width=0*, *height=0*, *fontsize=11*)

See the *Document()* constructor for details.

Returns

A document object.

This documentation covers all versions up to 1.28.2.

CHAPTER

FOURTEEN

CONVERTING FILES

14.1 Files to PDF

Document types supported by PyMuPDF can easily be converted to by using the *Document.save()* or *Document.ez_save()* method after they have been opened.

If you require a byte representation of the PDF, you can use *Document.convert_to_pdf()*. This method returns a buffer of data which can then be utilized by to create a new .

Example

```
import pymupdf

# Convert Markdown to PDF
md_doc = pymupdf.open("example.md")
md_doc.ez_save("example.pdf")

# Convert to PDF and get byte representation
pdf_bytes = md_doc.convert_to_pdf()

# Convert XPS to PDF
xps_doc = pymupdf.open("input.xps")
xps_doc.ez_save("output.pdf")
```

14.2 Markdown to PDF

14.2.1 Defining paper size

The default paper size is 400 x 600 *Rect* but you can specify a custom paper size if you wish, to do this just send through the *Rect* parameter as required, for example:

```
md_doc = pymupdf.open("example.md", rect=pymupdf.paper_rect("A4")) # A4 size
```

14.2.2 Defining CSS

By default, the Markdown content will be converted to PDF using a default CSS stylesheet. However, you can specify your own CSS stylesheet to customize the appearance of the resulting PDF. To do this, define your `css` and apply it.

For example, to make all `h1` headers red (The single # symbol in Markdown), you could do the following:

```
md_doc = pymupdf.open(  # open the Markdown document in A4 size
    "example.md",
    rect=pymupdf.paper_rect("A4")
)

css = "h1 {color:red;}"
md_doc.apply_css(css)

md_doc.ez_save("red-colored-header.pdf")
```

Note

The *support for CSS* is currently limited.

14.2.3 Using CSS Custom Element Selectors

Markdown input supports custom element selectors in the CSS, so you can define your own tags in the Markdown and then refer to them in the CSS.

In this way you can specify custom styles for specific elements in the Markdown content. For example, you could define a custom tag called `mytag` in the Markdown and then refer to it in the CSS to make it red:

```
css = """
mytag {
    color: red;
}
"""
```

And the corresponding Markdown:

```
# This is a header

This is some text.

<mytag>This text will be red.</mytag>
```

This is particularly useful for defining image sizes. For example, you could define a custom class called `my_image_class` in the CSS and then refer to it in the Markdown to style images:

```
css = """
my_image_class img {
    width: 100px;
    height: 100px;
}
"""
```

With the corresponding Markdown:

```
# This is a header

This is some text.

<my_image_class><img src="pie-chart.png" /></my_image_class>
```

14.2.4 Defining Fonts

Fonts can be defined by using the *Archive* parameter to provide a custom *Archive* containing the font files.

The fonts must exist in an archive which is provided to the *Archive* parameter when opening the Markdown file. The CSS can then refer to these fonts by their names as defined in the archive.

For example, assuming you have access to the source files for the "Comic Sans" font for all text, you could do the following:

```
# Global CSS instructions to use the "Comic Sans" font for all text. The font files must␣
↪be provided in the archive.
css = """
@font-face {font-family: sans-serif; src: url(comic.ttf);}
@font-face {font-family: sans-serif; src: url(comicbd.ttf); font-weight: bold;}
@font-face {font-family: sans-serif; src: url(comicz.ttf); font-weight: bold; font-
↪style: italic;}
@font-face {font-family: sans-serif; src: url(comici.ttf); font-style: italic;}
"""

archive = pymupdf.Archive("C:/Windows/Fonts")  # the fonts are here
archive.add(".")  # we've stored the archive image in this script's folder

md_file = "sample.md"
md_doc = pymupdf.open(  # open the Markdown document
    md_file,
    archive=archive,  # where to look for resources (fonts, images)
    rect=pymupdf.paper_rect("A4"),  # page dimension ISO A4
)

md_doc.apply_css(css)

md_doc.ez_save("sample.pdf")
```

14.3 PDF to Markdown

By utlilizing the *PyMuPDF4LLM API* we are able to convert PDF to a Markdown representation.

Example

```
import pymupdf4llm
import pathlib

md_text = pymupdf4llm.to_markdown("test.pdf")
print(md_text)

pathlib.Path("4llm-output.md").write_bytes(md_text.encode())
```

14.4 PDF to SVG

Technically, as SVG files cannot be multipage, we must export each page as an SVG.

To get an SVG representation of a page use the *Page.get_svg_image()* method.

Example

```
import pymupdf

doc = pymupdf.open("input.pdf")
page = doc[0]

# Convert page to SVG
svg_content = page.get_svg_image()

# Save to file
with open("output.svg", "w", encoding="utf-8") as f:
    f.write(svg_content)

doc.close()
```

14.5 PDF to Images

In order to convert a document to images, you should iterate the document pages, convert to *Pixmap* and save, e.g.:

```
doc = pymupdf.open("input.pdf")

for i, page in enumerate(doc):
    pix = page.get_pixmap(dpi=200)
    pix.save(f"page-{i+1}.png")

doc.close()
```

14.6 PDF to DOCX

Use the pdf2docx library which uses to provide document conversion from to **DOCX** format.

Example

```
from pdf2docx import Converter

pdf_file = 'input.pdf'
docx_file = 'output.docx'

# convert pdf to docx
cv = Converter(pdf_file)
cv.convert(docx_file) # all pages by default
cv.close()
```

This documentation covers all versions up to 1.28.2.

CHAPTER

FIFTEEN

COMPRESSING FILES

15.1 PyMuPDF & Compression

There are various ways to reduce the size of a PDF file and a variety of options for doing so with the *Document.save()* method parameters.

By using the mupdf_explored PDF as a control file let's look at the effect of various save-time parameters on the file size of a PDF document.

The original file is 1.8 MB in size. Let's see what happens when we save it with various optional parameters defined.

15.1.1 use_objstms

This boolean option packs object definitions into compressible streams.

Example

```
import pymupdf

doc = pymupdf.open("mupdf_explored.pdf")

doc.save(
    "output.pdf",
    use_objstms=True,  # pack object definitions into compressible streams
)
doc.close()
```

The result is a file size of 1.5 MB. However, we can achieve a better result by using the `deflate` parameter to compress the uncompressed streams.

15.1.2 deflate

By setting this option we can compress uncompressed streams.

Available options are:

Value	Meaning
0	No compression (this is the default)
1	Use standard Flate compression
2	Use Brotli compression (slowest, smallest output, but experimental and unsupported in many tools and viewers)[1]

Used in combination with `use_objstms` to pack object definitions into compressible streams we can achieve even better results.

Flate

Flate Example

```
import pymupdf

doc = pymupdf.open("mupdf_explored.pdf")

doc.save(
    "output.pdf",
    use_objstms=True,     # pack object definitions into compressible streams
    deflate=1,            # compress uncompressed streams with Flate compression
)
doc.close()
```

The result is a file size of 903 KB.

Note

Document.ez_save() automatically applies these parameters for you, so the same result can be achieved with:

```
doc.ez_save("output.pdf")
```

Brotli

Brotli Example

Warning

Brotli is experimental and unsupported in many tools and viewers.[1]

```
import pymupdf

doc = pymupdf.open("mupdf_explored.pdf")

doc.save(
    "output.pdf",
    use_objstms=True,     # pack object definitions into compressible streams
    deflate=2,            # compress uncompressed streams with Brotli compression
)
doc.close()
```

The result is a file size of 863 KB.

Brotli is expected to **yield meaningful differences** on text- and vector-heavy documents, and close to **none** on scanned or photo-heavy ones.

[1] If you've used Brotli to compress your PDFs and found that they don't work in your favorite viewer, please try MUPDF GL. If you still find a problem viewing the file then please file an issue on our Github tracker.

compression_effort

When using Brotli (deflate=2) by setting a further parameter compression_effort we can control how hard MuPDF works when compressing stream data. It trades CPU time for file size and *never changes what the document looks like*.

```
import pymupdf

doc = pymupdf.open("mupdf_explored.pdf")

doc.save(
    "output.pdf",
    use_objstms=True,     # pack object definitions into compressible streams
    deflate=2,            # compress uncompressed streams with Brotli compression
    compression_effort=100, # ask Brotli to work hard at it
)
doc.close()
```

The result now is a file size of 834 KB.

compression_effort: What it does and does not affect

It is an int, not a bool!

Value	Meaning
0	Zero effort (this is the default)
1	Minimum effort — fastest, largest output
100	Maximum effort — slowest, smallest output

This is the single most common mistake:

```
# WRONG - True == 1 == minimum effort
doc.save("out.pdf", use_objstms=True, deflate=2, compression_effort=True)

# CORRECT - maximum effort defined
doc.save("out.pdf", use_objstms=True, deflate=2, compression_effort=100)
```

Because Python's True is 1, passing a boolean silently selects an effort setting of 1. No warning is raised, and the file simply comes out larger than you expected.

Artifex published effort guidance in Brotli is Here! for the equivalent -e flag on mutool clean:

- **40** — roughly the speed of default Flate compression.
- **75** — roughly the speed of maximum Flate compression.
- **100** — squeezes out everything available, at a noticeably longer runtime.

That guidance is written in the context of Brotli compression, so treat the exact numbers as indicative.

compression_effort tunes the compressor applied to stream objects as the file is written. It does not decide *whether* streams are compressed using Brotli — that is deflate=2 — and it does not repack object definitions — that is use_objstms=True. Ensures that you have correctly set those two parameters before you start tuning compression_effort.

It has an effect on:

- content streams (text and vector graphics),
- font programs and other uncompressed binary streams,
- object streams produced by `use_objstms=True`.

It has little or no effect on:

- images already stored in a compressed format (JPEG, JPEG2000, JBIG2). Re-running a general-purpose compressor over them buys nothing. Use *Document.rewrite_images()* for those.
- documents whose bulk is unreferenced objects. Use `garbage=3` or `garbage=4`.

15.1.3 garbage

Documents with unreferenced objects can be reduced in size by setting the `garbage` parameter to 3 or 4. This will de-duplicate and drop unreferenced objects.

Value	Meaning
0	none (default)
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.

How much is saved here depends on the document, and how much "garbage" it may contain. For example, if we set `garbage=4` we only save a few KB:

```
import pymupdf

doc = pymupdf.open("mupdf_explored.pdf")

doc.save(
    "output.pdf",
    use_objstms=True,     # pack object definitions into compressible streams
    deflate=2,            # compress uncompressed streams with Brotli compression
    compression_effort=100, # ask Brotli to work hard at it
    garbage=4             # drop unreferenced objects, compact xref table, merge␣
↪duplicate objects, check streams for duplication
)
doc.close()
```

Now we have a resulting PDF with a file size of 831 KB.

15.2 Results Summary

Description	File size result	Comments
Original file	1.8 MB	
use_objstms=True	1.5 MB	
use_objstms=True, deflate=1	903 KB	
use_objstms=True, deflate=2	863 KB	Warning: Brotli is experimental and unsupported in many tools and viewers[Page 72, 1]
use_objstms=True, deflate=2, compression_effort=100	834 KB	Warning: Brotli is experimental and unsupported in many tools and viewers[Page 72, 1]
use_objstms=True, deflate=2, compression_effort=100, garbage=4	831 KB	Warning: Brotli is experimental and unsupported in many tools and viewers[Page 72, 1]

15.3 PyMuPDF & Compression Benchmarking

Gains are corpus-dependent. Measure before paying the CPU cost in a batch job:

```
import pathlib
import time

import pymupdf

src = "input.pdf"
for effort in (0, 40, 75, 100):
    doc = pymupdf.open(src)
    out = f"out_{effort}.pdf"
    start = time.perf_counter()
    doc.save(out, garbage=4, deflate=1, use_objstms=True,
             compression_effort=effort)
    elapsed = time.perf_counter() - start
    doc.close()
    size = pathlib.Path(out).stat().st_size
    print(f"effort={effort:>3}  {size/1e6:6.2f} MB  {elapsed:5.2f}s")
```

15.4 See also

- *Document.save()*
- *Document.ez_save()*
- *Document.rewrite_images()*
- *Document.subset_fonts()*
- Brotli is Here!

This documentation covers all versions up to 1.28.2.

CHAPTER

SIXTEEN

OCR - OPTICAL CHARACTER RECOGNITION

has integrated support for OCR (Optical Character Recognition). It is possible to use OCR for both, images (via the *Pixmap* class) and for document pages.

The feature is currently based on Tesseract-OCR which must be installed as a separate application – see the *Enabling Integrated OCR Support*.

16.1 How to OCR an Image

A supported image must first be converted to a *Pixmap*. The Pixmap can then be saved to a 1-page PDF. This page will look like the original image with the same width and height. It will contain a layer of text as recognized by Tesseract.

The PDF can be generated via one of the methods *Pixmap.pdfocr_save()* or *Pixmap.pdfocr_tobytes()*, as a file on disk or as a PDF in memory.

The text can be extracted and searched with the usual text extraction and search methods (*Page.get_text()*, *Page.search_for()*, etc.). Please also note the following important facts and prerequisites:

- When converting the image to a Pixmap, please confirm that the color space is RGB and alpha is `False` (no transparency). Convert the original Pixmap if necessary.
- All text is written as "hidden" with Tesseract's own `GlyphLessFont`, a mono-spaced font with metrics comparable to Courier.
- All text has the properties regular and black (i.e. no bold, no italic, no information about the original fonts).
- Tesseract does not recognize vector graphics (i.e. no drawings / line-art).

This approach is also recommended to OCR a complete scanned PDF:

- Render each page to a *Pixmap* with desired resolution
- Append the resulting 1-page PDF to the output PDF

16.2 How to OCR a Document Page

Any supported document page can be OCR-ed – either the complete page or only the image areas on it.

Because optical character recognition is about one thousand times slower than standard text extraction, we make sure to do OCR only once per page and store the result in a *TextPage*. Using this TextPage for all subsequent extractions and text searches will then happen with 's usual top speed.

To OCR a document page, follow this approach:

1. Determine whether OCR is needed / beneficial at all. A number of criteria can be used for this decision, like:

- page is completely covered by an image

- no text exists on the page
- thousands of small vector graphics (indicating *simulated* text)

2. OCR the page and store result in a *TextPage* object using an instruction like `tp = page.get_textpage_ocr(...)`.
3. Refer to the produced *TextPage* in all subsequent text extractions and searches via the `textpage=tp` parameter.

This documentation covers all versions up to 1.28.2.

CHAPTER

SEVENTEEN

TEXT

17.1 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*.

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, pathlib, pymupdf
fname = sys.argv[1]  # get document filename
with pymupdf.open(fname) as doc:  # open document
    text = chr(12).join([page.get_text() for page in doc])
# write as a binary file to support non-ASCII characters
pathlib.Path(fname + ".txt").write_bytes(text.encode())
```

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.

17.2 How to Extract Text as Markdown

This is especially useful for *RAG/LLM* environments - please see *Outputting as Markdown*.

17.3 How to Extract Key-Value Pairs from a Page

If the layout of a page is *"predictable"* in some sense, then there is a simple way to find the values for a given set of keywords fast and easily – without using regular expressions. Please see this example script.

"Predictable" in this context means:

- Every keyword is followed by its value – no other text is present in between them.

- The bottom of the value's boundary box is **not above** the one of the keyword.
- There are **no other restrictions**: the page layout may or may not be fixed, and the text may also have been stored as one string. Key and value may have any distance from each other.

For example, the following five key-value pairs will be correctly identified:

```
key1                value1
key2
value2
key3
       value3 blah, blah, blah key4 value4 some other text key5 value5 ...
```

17.4 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.

17.5 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 = pymupdf.open("some.pdf")
header = "Header"  # text in header
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),
        f"Page {page.number + 1} of {doc.page_count}", # text in footer
    )
```

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 `pymupdf` module in CLI: `python -m pymupdf 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.

17.6 How to Extract Table Content from Documents

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

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

This can be a very complex task, depending on details like the presence or absence of lines, rectangles or other supporting vector graphics.

Method *Page.find_tables()* does all that for you, with a high table detection precision. Its great advantage is that there are no external library dependencies, nor the need to employ artificial intelligence or machine learning technologies. It also provides an integrated interface to the well-known Python package for data analysis pandas.

Please have a look at example Jupyter notebooks, which cover standard situations like multiple tables on one page or joining table fragments across multiple pages.

17.7 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 recommended, when page rotation is not zero.

But you also have other options:

```
import sys
import pymupdf

def mark_word(page, text):
    """Underline each word that contains 'text'.
    """
    found = 0
    wlist = page.get_text("words", delimiters=None)  # 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 = pymupdf.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 = pymupdf.open(fname)
```

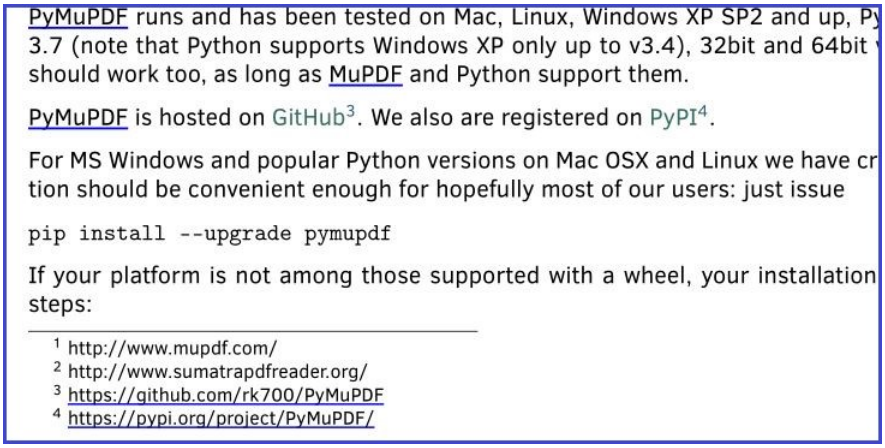
(continues on next page)

(continued from previous page)

```
print(f"underlining words containing '{word}' in document '{doc.name}'")

new_doc = False  # indicator if anything found at all

for page in doc:  # scan through the pages
    found = mark_word(page, text)  # mark the page's words
    if found:  # if anything found ...
        new_doc = True
        print(f"found '{text}' {found} times on page {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 white spaces 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 word delimiters**. By default, word delimiters are white spaces and the non-breaking space `chr(0xA0)`. If you use extra delimiting characters like `page.get_text("words", delimiters="./,")` then none of these characters should be included in your search string either.
- As shown here, upper / lower cases are **respected**. But this can be changed by using the string method *lower()* (or even regular expressions) in function *mark_word*.
- There is **no upper limit**: all occurrences will be detected.
- You can use **anything** to mark the word: 'Underline', 'Highlight', 'StrikeThrough' or 'Square' annotations, etc.
- Here is an example snippet of a page of this manual, where "MuPDF" has been used as the search string. Note that all strings **containing "MuPDF"** have been completely underlined (not just the search string).


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

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

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

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

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

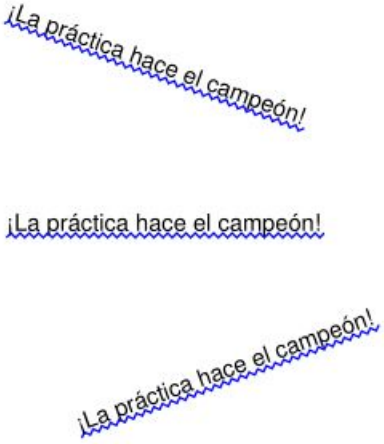
[1] http://www.mupdf.com/
[2] http://www.sumatrapdfreader.org/
[3] https://github.com/rk700/PyMuPDF
[4] https://pypi.org/project/PyMuPDF/

17.8 How to Mark Searched Text

This script searches for text and marks it:

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

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

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

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

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



17.9 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 != 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 = pymupdf.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 = pymupdf.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.

17.10 How to Analyze Font Characteristics

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

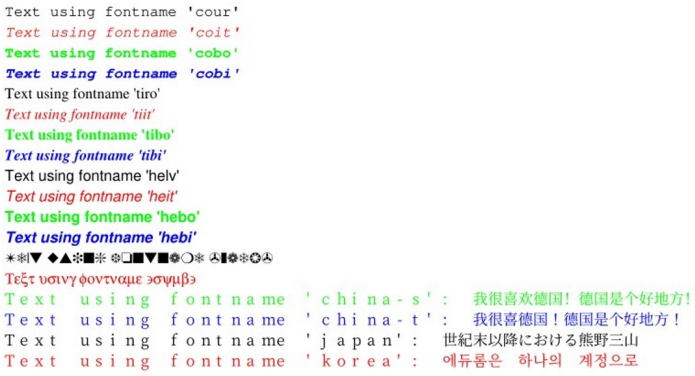
```
import sys

import pymupdf


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

(continues on next page)

(continued from previous page)

```
    return ", ".join(l)


doc = pymupdf.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
    for l in b["lines"]:  # iterate through the text lines
        for s in l["spans"]:  # iterate through the text spans
            print("")
            s_font = s['font']
            s_flags = flags_decomposer(s['flags'])
            s_size = s['size']
            s_color = s['color']
            print(f"Text: '{s['text']}'")  # simple print of text
            print(f"Font: '{s_font}' ({s_flags}), size {s_size}, color #{s_color:06x}")
```

Here is the PDF page and the script output:




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

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

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

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

17.11 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.

17.11.1 How to Write Text Lines

Output some text lines on a page:

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

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

rc = page.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(f"{rc} lines printed on page {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.

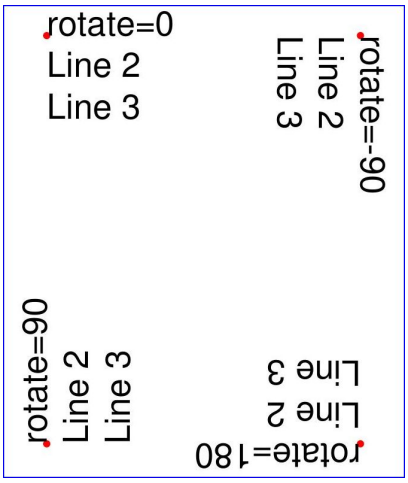
Line **width is ignored**. The surplus part of a line will simply be invisible.

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 pymupdf
doc = pymupdf.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 = pymupdf.Point(25, 25)
p2 = pymupdf.Point(page.rect.width - 25, 25)
p3 = pymupdf.Point(25, page.rect.height - 25)
p4 = pymupdf.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:



17.11.2 How to Fill a Text Box

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

```
import pymupdf

doc = pymupdf.open()  # new or existing PDF
page = doc.new_page()  # new page, or choose doc[n]

# write in this overall area
rect = pymupdf.Rect(100, 100, 300, 150)

# partition the area in 4 equal sub-rectangles
CELLS = pymupdf.make_table(rect, cols=4, rows=1)

t1 = "text with rotate = 0."  # these texts we will written
t2 = "text with rotate = 90."
t3 = "text with rotate = 180."
t4 = "text with rotate = 270."
text = [t1, t2, t3, t4]
red = pymupdf.pdfcolor["red"]  # some colors
gold = pymupdf.pdfcolor["gold"]
blue = pymupdf.pdfcolor["blue"]
"""
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
for i in range(len(CELLS[0])):
    shape.draw_rect(CELLS[0][i])  # draw rectangle
    shape.insert_textbox(
        CELLS[0][i], text[i], fontname="hebo", color=blue, rotate=90 * i
    )

shape.finish(width=0.3, color=red, fill=gold)
```

(continues on next page)

(continued from previous page)

```
shape.commit()  # write all stuff to the page
doc.ez_save(__file__.replace(".py", ".pdf"))
```

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

17.11.3 How to Fill a Box with HTML Text

Method *Page.insert_htmlbox()* offers a **much more powerful** way to insert text in a rectangle.

Instead of simple, plain text, this method accepts HTML source, which may not only contain HTML tags but also styling instructions to influence things like font, font weight (bold) and style (italic), color and much more.

It is also possible to mix multiple fonts and languages, to output HTML tables and to insert images and URI links.

For even more styling flexibility, an additional CSS source may also be given.

The method is based on the *Story* class. Therefore, complex script systems like Devanagari, Nepali, Tamil and many are supported and written correctly thanks to using the HarfBuzz library - which provides this so-called **“text shaping”** feature.

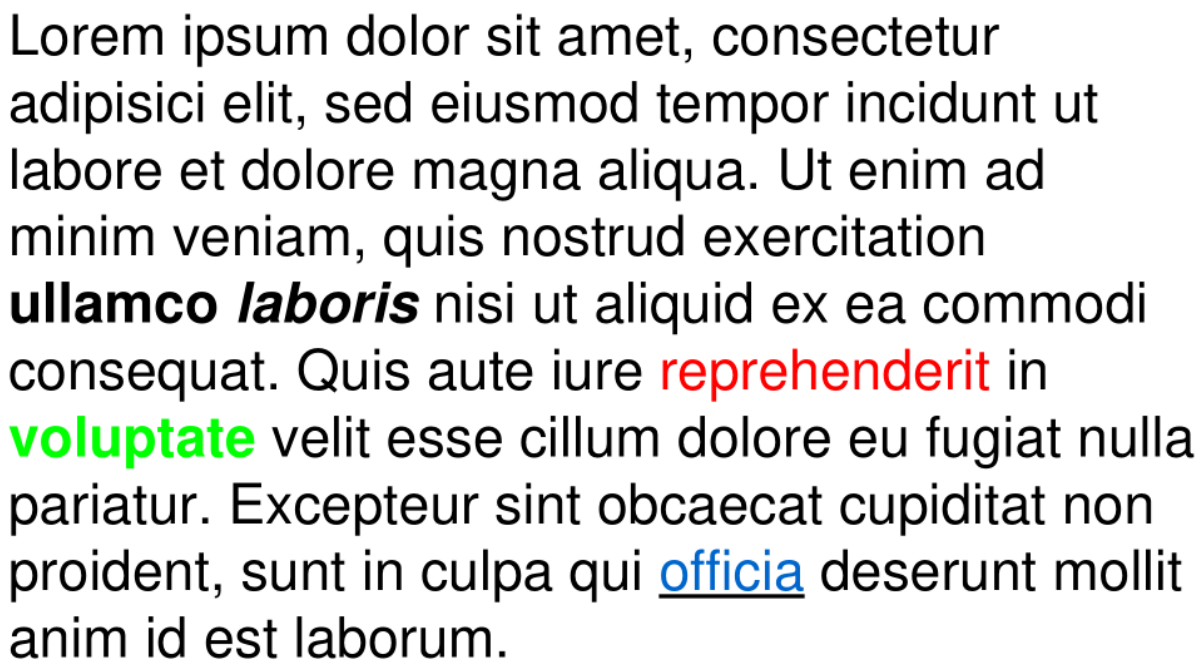
Any required fonts to output characters are automatically pulled in from the Google NOTO font library - as a fallback (when the – optionally supplied – user font(s) do not contain some glyphs).

As a small glimpse into the features offered here, we will output the following HTML-enriched text:

```
import pymupdf


rect = pymupdf.Rect(100, 100, 400, 300)

text = """Lorem ipsum dolor sit amet, consectetur adipisici elit, sed
    eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad
    minim veniam, quis nostrud exercitation <b>ullamco <i>laboris</i></b>
    nisi ut aliquid ex ea commodi consequat. Quis aute iure
    <span style="color: #f00;">reprehenderit</span>
    in <span style="color: #0f0;font-weight:bold;">voluptate</span> velit
    esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat
    cupiditat non proident, sunt in culpa qui
    <a href="https://www.artifex.com">officia</a> deserunt mollit anim id
    est laborum."""

doc = pymupdf.Document()

page = doc.new_page()
page.insert_htmlbox(rect, text, css="* {font-family: sans-serif;font-size:14px;}")

doc.ez_save(__file__.replace(".py", ".pdf"))
```

Please note how the “css” parameter is used to globally select the default “sans-serif” font and a font size of 14.

The result will look like this:



How to output HTML tables and images

Here is another example that outputs a table with this method. This time, we are including all the styling in the HTML source itself. Please also note, how it works to include an image - even within a table cell:

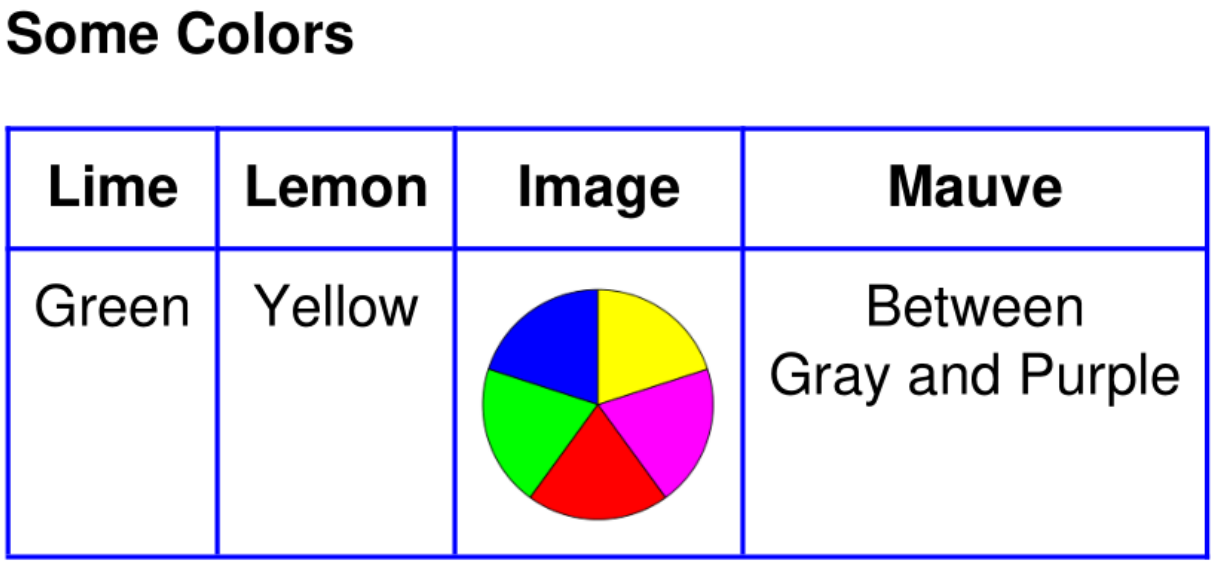
```
import pymupdf
import os

filedir = os.path.dirname(__file__)


text = """
<style>
body {
    font-family: sans-serif;
}

td,
th {
    border: 1px solid blue;
    border-right: none;
    border-bottom: none;
    padding: 5px;
    text-align: center;
}

table {
    border-right: 1px solid blue;
    border-bottom: 1px solid blue;
```

(continues on next page)

(continued from previous page)

```
    border-spacing: 0;
}
</style>

<body>
<p><b>Some Colors</b></p>
<table>
    <tr>
    <th>Lime</th>
    <th>Lemon</th>
    <th>Image</th>
    <th>Mauve</th>
    </tr>
    <tr>
    <td>Green</td>
    <td>Yellow</td>
    <td><img src="img-cake.png" width=50></td>
    <td>Between<br>Gray and Purple</td>
    </tr>
</table>
</body>
"""

doc = pymupdf.Document()

page = doc.new_page()
rect = page.rect + (36, 36, -36, -36)

# we must specify an Archive because of the image
page.insert_htmlbox(rect, text, archive=pymupdf.Archive("."))

doc.ez_save(__file__.replace(".py", ".pdf"))
```

The result will look like this:

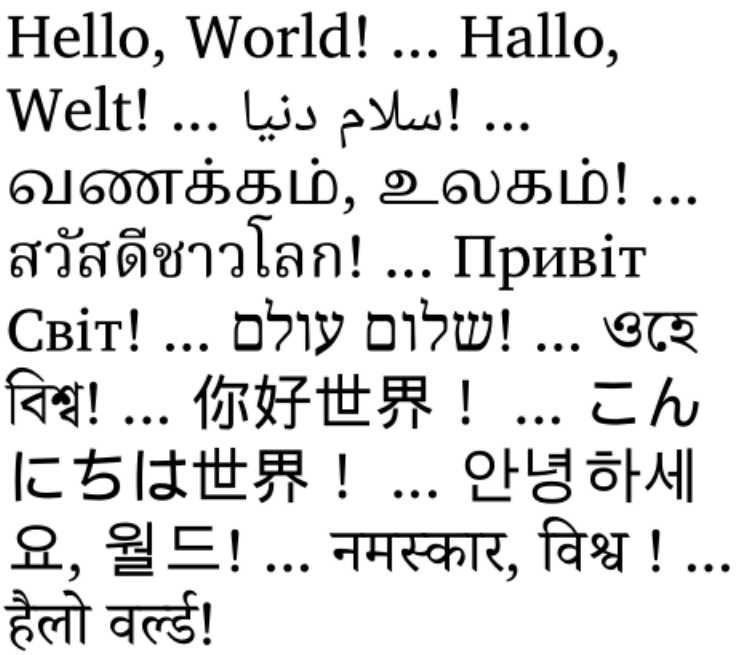


How to Output Languages of the World

Our third example will demonstrate the automatic multi-language support. It includes automatic **text shaping** for complex scripting systems like Devanagari and right-to-left languages:

```
import pymupdf

greetings = (
    "Hello, World!",  # english
    "Hallo, Welt!",  # german
    " !",  # persian
    ", !",  # tamil
    "!",  # thai
    " !",  # ucranian
    " !",  # hebrew
    " !",  # bengali
    "",  # chinese
    "",  # japanese
    ", !",  # korean
    ",  !",  # sanskrit
    " !",  # hindi
)
doc = pymupdf.open()
page = doc.new_page()
rect = (50, 50, 200, 500)

# join greetings into one text string
text = " ... ".join([t for t in greetings])

# the output of the above is simple:
page.insert_htmlbox(rect, text)
doc.save(__file__.replace(".py", ".pdf"))
```

And this is the output:



How to Specify your Own Fonts

Define your font files in CSS syntax using the `@font-face` statement. You need a separate `@font-face` for every combination of font weight and font style (e.g. bold or italic) you want to be supported. The following example uses the famous MS Comic Sans font in its four variants regular, bold, italic and bold-italic.

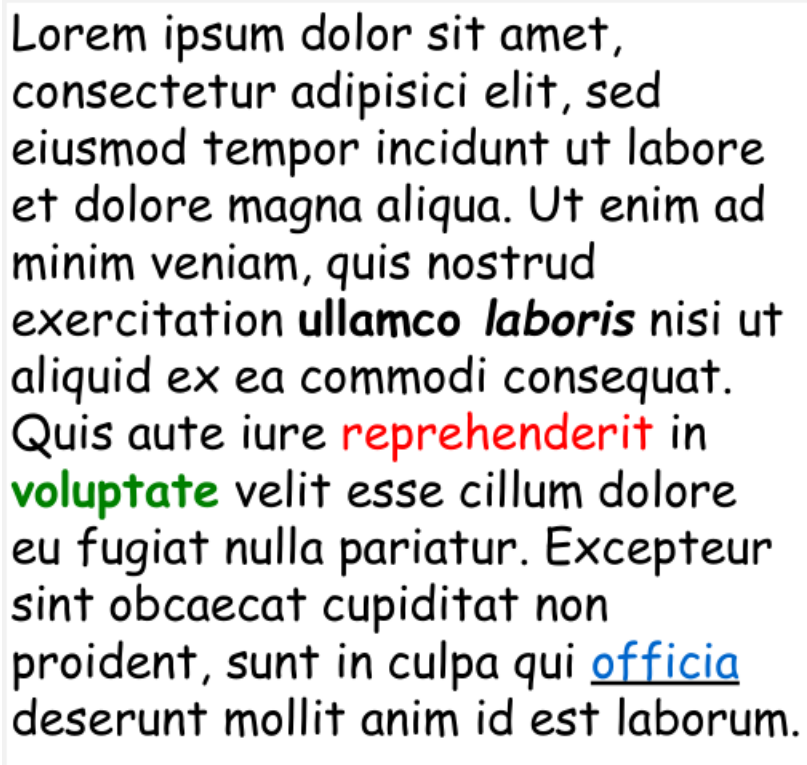
As these four font files are located in the system's folder `C:/Windows/Fonts` the method needs an *Archive* definition that points to that folder:

```
"""
How to use your own fonts with method Page.insert_htmlbox().
"""
import pymupdf

# Example text
text = """Lorem ipsum dolor sit amet, consectetur adipisici elit, sed
    eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad
    minim veniam, quis nostrud exercitation <b>ullamco <i>laboris</i></b>
    nisi ut aliquid ex ea commodi consequat. Quis aute iure
    <span style="color: red;">reprehenderit</span>
    in <span style="color: green;font-weight:bold;">voluptate</span> velit
    esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat
    cupiditat non proident, sunt in culpa qui
    <a href="https://www.artifex.com">officia</a> deserunt mollit anim id
    est laborum."""

"""
We need an Archive object to show where font files are located.
We intend to use the font family "MS Comic Sans".
"""
arch = pymupdf.Archive("C:/Windows/Fonts")
```

(continues on next page)

(continued from previous page)

```
# These statements define which font file to use for regular, bold,
# italic and bold-italic text.
# We assign an arbitrary common font-family for all 4 font files.
# The Story algorithm will select the right file as required.
# We request to use "comic" throughout the text.
css = """
@font-face {font-family: comic; src: url(comic.ttf);}
@font-face {font-family: comic; src: url(comicbd.ttf);font-weight: bold;}
@font-face {font-family: comic; src: url(comicz.ttf);font-weight: bold;font-style:␣
↪italic;}
@font-face {font-family: comic; src: url(comici.ttf);font-style: italic;}
* {font-family: comic;}
"""

doc = pymupdf.Document()
page = doc.new_page(width=150, height=150)  # make small page

page.insert_htmlbox(page.rect, text, css=css, archive=arch)

doc.subset_fonts(verbose=True)  # build subset fonts to reduce file size
doc.ez_save(__file__.replace(".py", ".pdf"))
```



How to Request Text Alignment

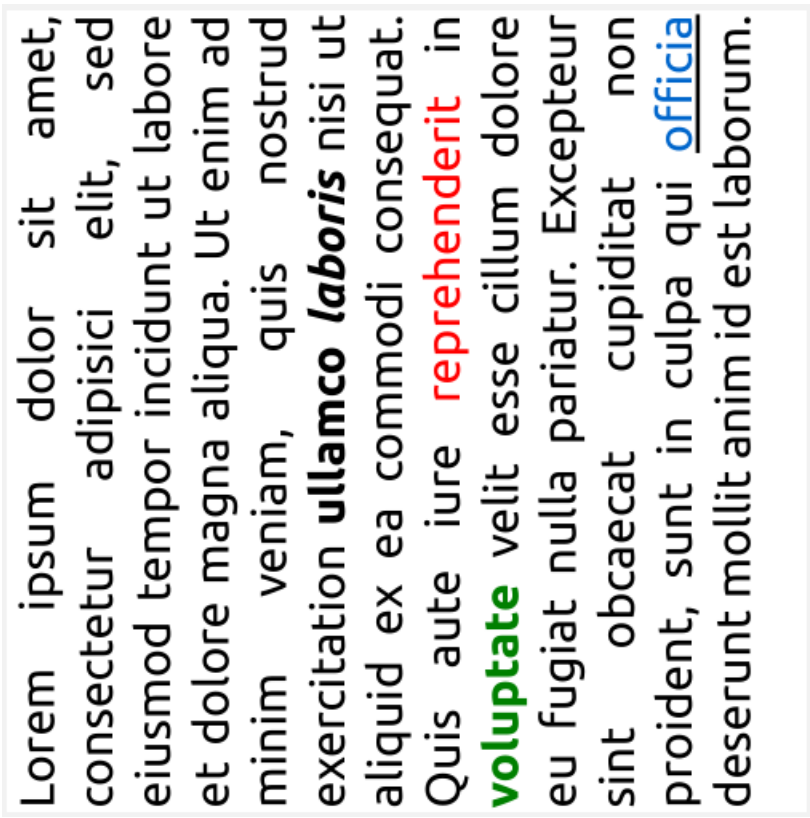
This example combines multiple requirements:

- Rotate the text by 90 degrees anti-clockwise.
- Use a font from package pymupdf-fonts. You will see that the respective CSS definitions are a lot easier in this

case.

- Align the text with the "justify" option.

```
"""
How to use a pymupdf font with method Page.insert_htmlbox().
"""
import pymupdf

# Example text
text = """Lorem ipsum dolor sit amet, consectetur adipisici elit, sed
    eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad
    minim veniam, quis nostrud exercitation <b>ullamco <i>laboris</i></b>
    nisi ut aliquid ex ea commodi consequat. Quis aute iure
    <span style="color: red;">reprehenderit</span>
    in <span style="color: green;font-weight:bold;">voluptate</span> velit
    esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat
    cupiditat non proident, sunt in culpa qui
    <a href="https://www.artifex.com">officia</a> deserunt mollit anim id
    est laborum."""

"""
This is similar to font file support. However, we can use a convenience
function for creating required CSS definitions.
We still need an Archive for finding the font binaries.
"""
arch = pymupdf.Archive()

# We request to use "myfont" throughout the text.
css = pymupdf.css_for_pymupdf_font("ubuntu", archive=arch, name="myfont")
css += "* {font-family: myfont;text-align: justify;}"

doc = pymupdf.Document()

page = doc.new_page(width=150, height=150)

page.insert_htmlbox(page.rect, text, css=css, archive=arch, rotate=90)

doc.subset_fonts(verbose=True)
doc.ez_save(__file__.replace(".py", ".pdf"))
```



17.12 How to Extract Text with Color

Iterate through your text blocks and find the spans of text you need for this information.

```
for page in doc:
    text_blocks = page.get_text("dict", flags=pymupdf.TEXTFLAGS_TEXT)["blocks"]
    for block in text_blocks:
        for line in block["lines"]:
            for span in line["spans"]:
                text = span["text"]
                color = pymupdf.sRGB_to_rgb(span["color"])
                print(f"Text: {text}, Color: {color}")
```

This documentation covers all versions up to 1.28.2.

CHAPTER

EIGHTEEN

IMAGES

18.1 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*.

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, pymupdf  # import the bindings
fname = sys.argv[1]  # get filename from command line
doc = pymupdf.open(fname)  # open document
for page in doc:  # iterate through the pages
    pix = page.get_pixmap()  # render page to an image
    pix.save(f"page-{page.number}.png")  # 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.

18.2 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 = pymupdf.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

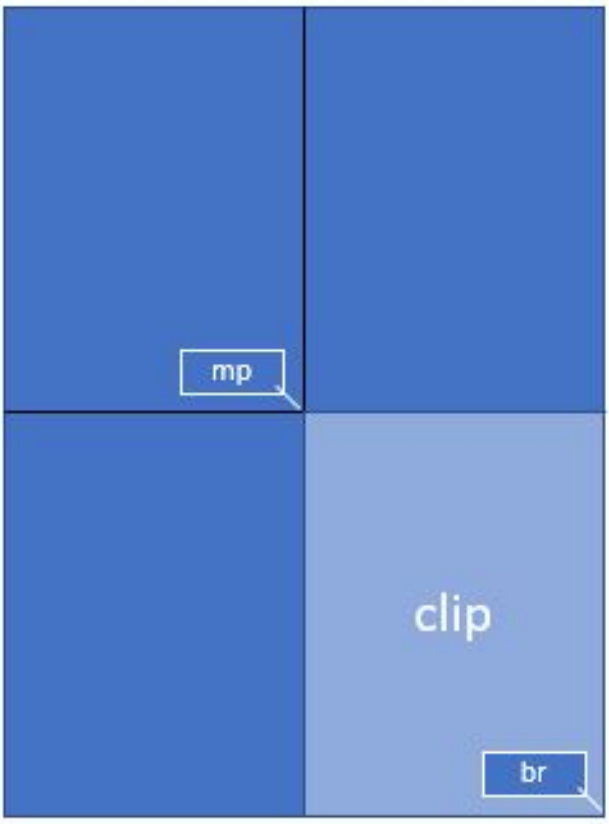
notation brevity, this approach has the additional advantage that the **dpi value is saved with the image** file – which does not happen automatically when using the Matrix notation.

18.3 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 = pymupdf.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 = pymupdf.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*.

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

18.5 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.

18.6 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 = pymupdf.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}
```

18.7 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 = pymupdf.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 = pymupdf.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:

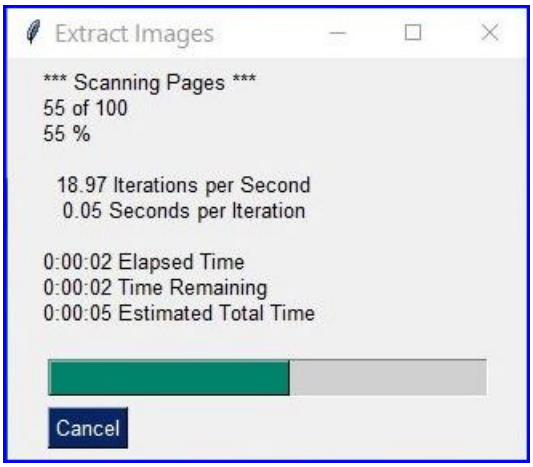
a. **"Inspect the page objects:"** Loop through the items of *Page.get_images()*. It is a list of list, and its items look like *[xref, smask, ...]*, containing the *xref* of an image. This *xref* can then be used with one of the above methods. Use this method for **valid (undamaged)** documents. Be 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.

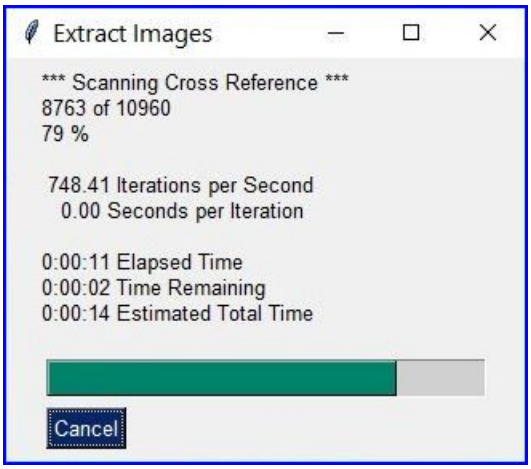
Use this method if the PDF is **damaged (unusable pages)**. Note that a PDF often contains "pseudo-images" ("stencil masks") with the special purpose of defining the transparency of some other image. You may want to provide logic to exclude those from extraction. Also have a look at the next section.

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

extract-from-pages.py extracts images page by page:



and extract-from-xref.py extracts images by xref table:



18.8 How to Handle Image Masks

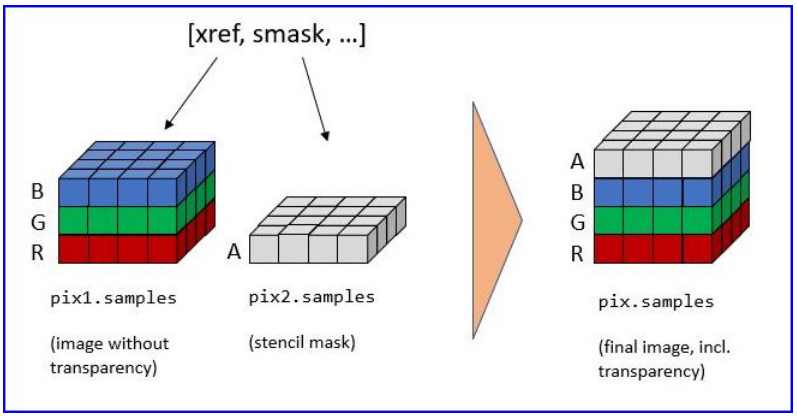
Some images in PDFs are accompanied by **image masks**. In their simplest form, masks represent alpha (transparency) bytes stored as separate images. In order to reconstruct the original of an image, which has a mask, it must be "enriched" with transparency bytes taken from its mask.

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 = pymupdf.Pixmap(doc.extract_image(xref)["image"])    # (1) pixmap of image w/o␣
↪alpha
>>> mask = pymupdf.Pixmap(doc.extract_image(smask)["image"])   # (2) mask pixmap
>>> pix = pymupdf.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-from-pages.py, and extract-from-xref.py above also contain this logic.

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

for i, f in enumerate(imglist):
    img = pymupdf.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 = pymupdf.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
```

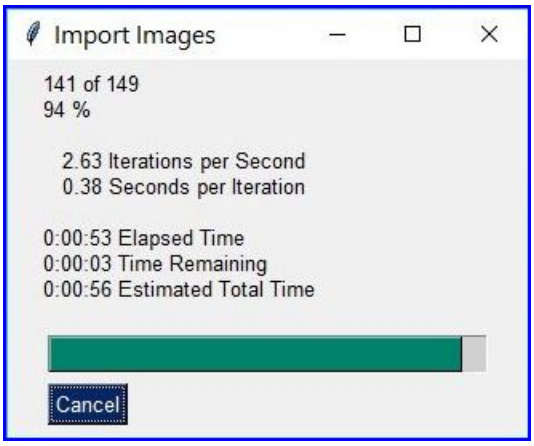
(continues on next page)

(continued from previous 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

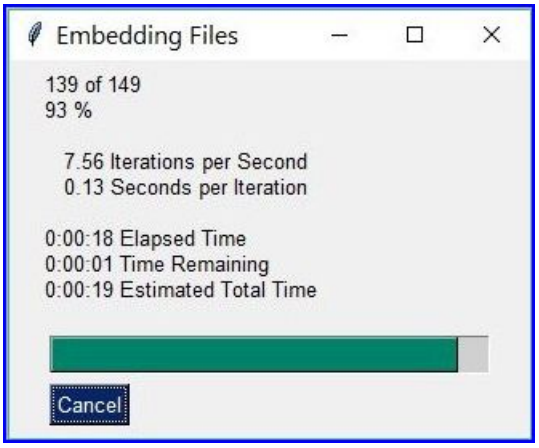
The second script **embeds** arbitrary files – not only images. The resulting PDF will have just one (empty) page, required for technical reasons. To later access the embedded files again, you would need a suitable PDF viewer that can display and / or extract embedded files:

```
import os, pymupdf
import PySimpleGUI as psg  # for showing progress bar
doc = pymupdf.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.embfile_add(img, f, filename=f,  # and embed it
```

(continues on next page)

(continued from previous page)

```
                        ufilename=f, desc=f)
    psg.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")
```

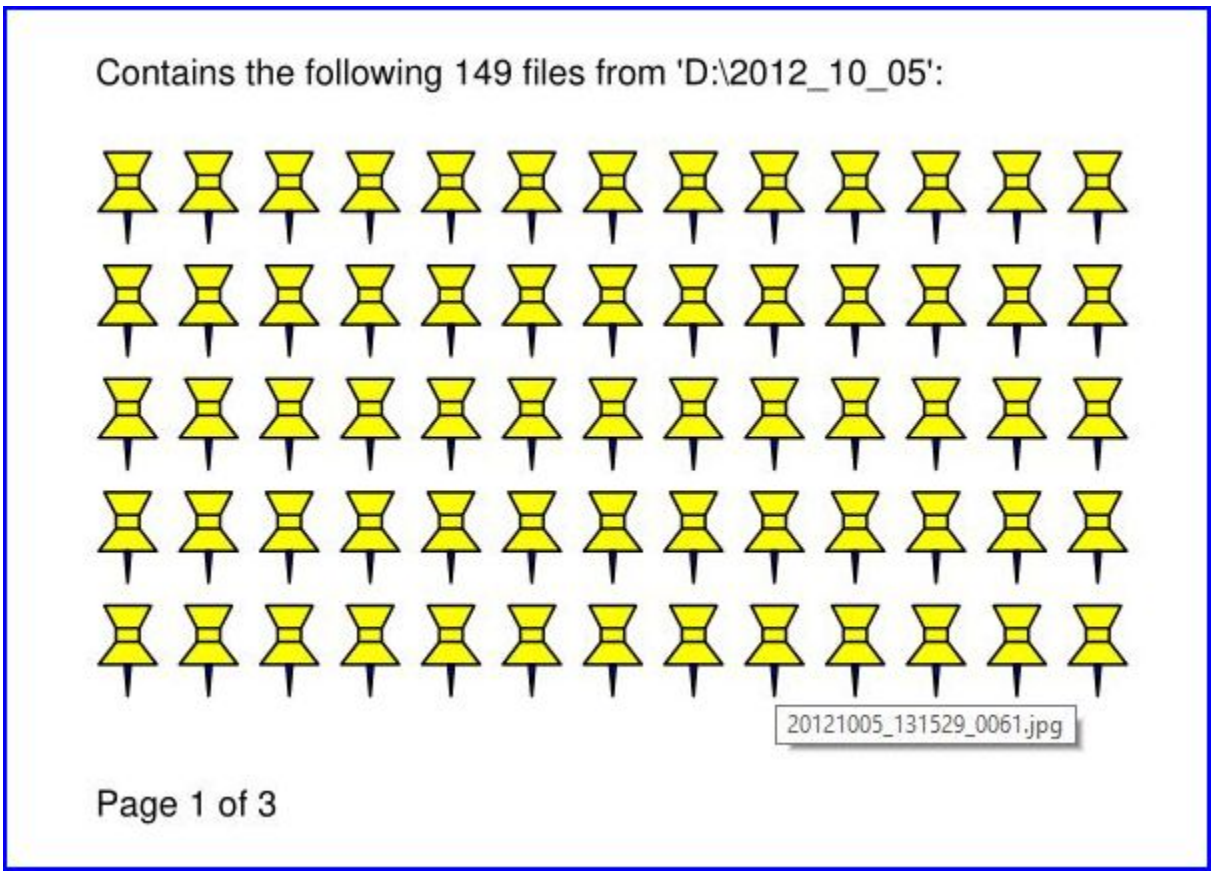


This is by far the fastest method, and it also produces the smallest possible output file size. The above pictures needed 20 seconds on my machine and yielded a PDF size of 510 MB. Look here for a more complete source code: it offers a directory selection dialog and skips non-file entries.

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!

18.10 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=pymupdf.Identity)* delivers a UTF-8 string *svg* which can be stored with extension ".svg".

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

Remarks

1. The **input** argument of *pymupdf.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 = pymupdf.Pixmap("myfamily.jpg")
pix.save("myfamily.psd")
```

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 = pymupdf.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:

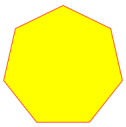
```
import tkinter as tk
pix = pymupdf.Pixmap("input.png")  # may have an alpha channel
if pix.alpha:  # we have an alpha channel!
    pix = pymupdf.Pixmap(pix, 0)  # remove it
tkimg = tk.PhotoImage(data=pix.tobytes("ppm"))
```

18.12 How to Use Pixmaps: Gluing Images

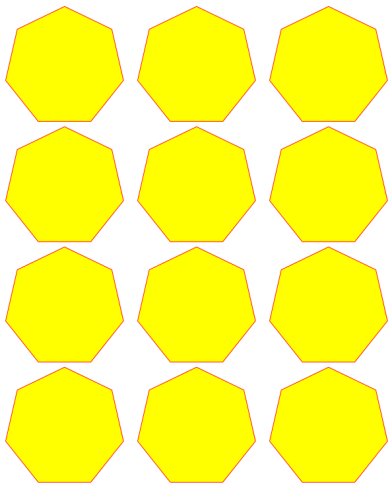
This shows how pixmaps can be used for purely graphical, non-document purposes. The script reads an image file and creates a new image which consist of 3 * 4 tiles of the original:

```
import pymupdf
src = pymupdf.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 = pymupdf.Pixmap(src.colorspace, (0, 0, tar_w, tar_h), src.alpha)

# now fill target with the tiles
for i in range(col):
    for j in range(lin):
        src.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:



18.13 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-suqares (3 times 3) and cut out the one in the center. Treat each of the remaining eight sub-squares in the same way, and continue *ad infinitum*. The end result is a set with area zero and fractal dimension 1.8928...

This script creates 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):

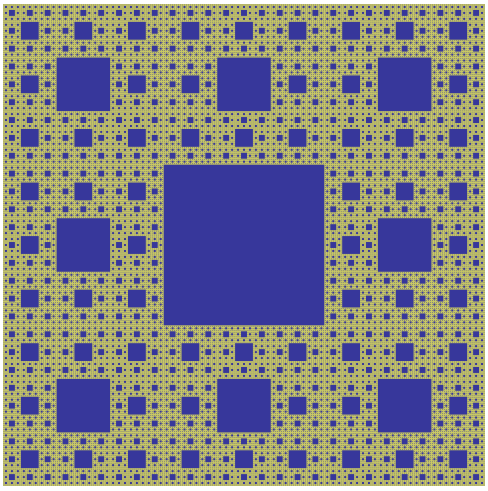
```
import pymupdf, time
if not list(map(int, pymupdf.VersionBind.split("."))) >= [1, 14, 8]:
    raise SystemExit("need PyMuPDF v1.14.8 for this script")
n = 6                           # depth (precision)
d = 3**n                        # edge length

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

pm = pymupdf.Pixmap(pymupdf.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 = pymupdf.Pixmap(pymupdf.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 (f"{round(t1-t0,3)} sec to create / fill the pixmap")
print (f"{round(t2-t1,3)} sec to save the image")
```

The result should look something like this:



18.14 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 pymupdf
```

(continues on next page)

(continued from previous page)

```
#==============================================================================
# create a fun-colored width * height PNG with pymupdf 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 = pymupdf.Pixmap(pymupdf.csRGB, width, height, samples, alpha=False)
pix.save("test.png")
```

18.15 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	*Page.insert_image()*	*Page.show_pdf_page()*
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 *Document.convert_to_pdf()*

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)
```

(continues on next page)

(continued from previous page)

```
    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
)
```

18.16 How to Use Pixmaps: Checking Text Visibility

Whether or not a given piece of text is actually visible on a page depends on a number of factors:

1. Text is not covered by another object but may have the same color as the background i.e., white-on-white etc.
2. Text may be covered by an image or vector graphics. Detecting this is an important capability, for example to uncover badly anonymized legal documents.
3. Text is created hidden. This technique is usually used by OCR tools to store the recognized text in an invisible layer on the page.

The following shows how to detect situation 1. above, or situation 2. if the covering object is unicolor:

```
pix = page.get_pixmap(dpi=150)  # make page image with a decent resolution

# the following matrix transforms page to pixmap coordinates
mat = page.rect.torect(pix.irect)

# search for some string "needle"
rlist = page.search_for("needle")
# check the visibility for each hit rectangle
for rect in rlist:
    if pix.color_topusage(clip=rect * mat)[0] > 0.95:
        print("'needle' is invisible here:", rect)
```

Method *Pixmap.color_topusage()* returns a tuple `(ratio, pixel)` where 0 < ratio <= 1 and *pixel* is the pixel value of the color. Please note that we create a **pixmap only once**. This can save a lot of processing time if there are multiple hit rectangles.

The logic of the above code is: If the needle's rectangle is ("almost": > 95%) unicolor, then the text cannot be visible. A typical result for visible text returns the color of the background (mostly white) and a ratio around 0.7 to 0.8, for example `(0.685, b'xffxffxff')`.

This documentation covers all versions up to 1.28.2.

CHAPTER

NINETEEN

ANNOTATIONS

19.1 How to Add and Modify Annotations

In , 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.

Annotations can **only** be inserted in pages - other document types do not support annotation insertion.

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

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

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

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

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

import gc
import sys

import pymupdf

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

(continues on next page)

(continued from previous page)

```
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 = pymupdf.Rect(0, 50, 0, 50)
r = pymupdf.Rect(72, 72, 220, 100)
t1 = u"têxt üsès Lätiñ 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), f"{annot.type[1]} annotation", color=red)


doc = pymupdf.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=pymupdf.TEXT_ALIGN_CENTER,
)
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(
```

(continues on next page)

(continued from previous page)

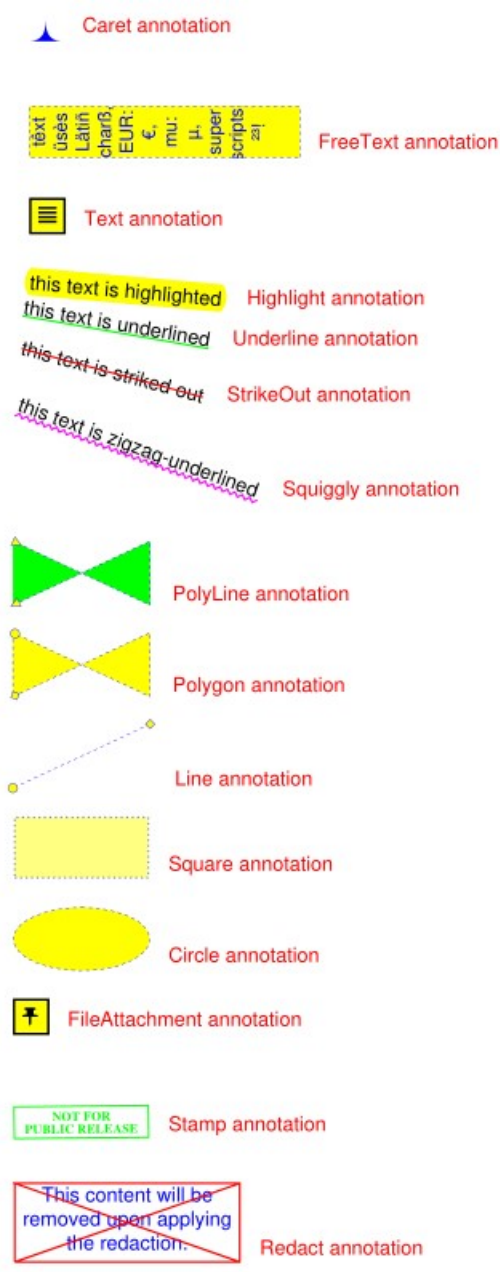
```
    pos,  # insertion point
    highlight,  # inserted text
    morph=(pos, pymupdf.Matrix(-5)),  # rotate around insertion point
)
rl = page.search_for(highlight, quads=True)  # need a quad b/o tilted text
annot = page.add_highlight_annot(rl[0])
print_descr(annot)

pos = annot.rect.bl  # next insertion point
page.insert_text(pos, underline, morph=(pos, pymupdf.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, pymupdf.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, pymupdf.Matrix(-20)))
rl = page.search_for(squiggled, quads=True)
annot = page.add_squiggly_annot(rl[0])
print_descr(annot)

pos = annot.rect.bl
r = pymupdf.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(pymupdf.PDF_ANNOT_LE_CLOSED_ARROW, pymupdf.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)
annot.set_line_ends(pymupdf.PDF_ANNOT_LE_DIAMOND, pymupdf.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(pymupdf.PDF_ANNOT_LE_DIAMOND, pymupdf.PDF_ANNOT_LE_CIRCLE)
annot.update()
print_descr(annot)
```

(continues on next page)

(continued from previous page)

```
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=pymupdf.TEXT_ALIGN_CENTER,
)
annot = page.add_redact_annot(r)
print_descr(annot)

doc.save(__file__.replace(".py", f"-{page.rotation:d}.pdf"), deflate=True)
```

This script should lead to the following output:



19.2 How to Use FreeText

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

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

# some colors
blue = (0, 0, 1)
```

(continues on next page)

(continued from previous page)

```
green = (0, 1, 0)
red = (1, 0, 0)
gold = (1, 1, 0)

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

# 3 rectangles, same size, above each other
r1 = pymupdf.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, text_color=red)
a2 = page.add_freetext_annot(r2, t, fontname="Ti", text_color=blue)
a3 = page.add_freetext_annot(r3, t, fontname="Co", text_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:

Here is an example for using rich text and call-out lines:

```
import pymupdf

"""Use rich text for FreeText annotations"""
```

(continues on next page)

(continued from previous page)

```
# define an overall styling
ds = """font-size: 11pt; font-family: sans-serif;"""

# some special characters
bullet = chr(0x2610) + chr(0x2611) + chr(0x2612)

# the annotation text with HTML and styling syntax
text = f"""<p style="text-align:justify;margin-top:-25px;">
PyMuPDF <span style="color: red;">  </span>
<span style="color:blue;">Here is some <b>bold</b> and <i>italic</i> text, followed by
↪<b><i>bold-italic</i></b>. Text-based check boxes: {bullet}.</span>
 </p>"""

# here are some colors
gold = (1, 1, 0)
green = (0, 1, 0)

# new/empty PDF
doc = pymupdf.open()

# make a page in ISO-A4 format
page = doc.new_page()

# text goes into this:
rect = pymupdf.Rect(100, 100, 350, 200)

# define some points for callout lines
p2 = rect.tr + (50, 30)
p3 = p2 + (0, 30)

# define the annotation
annot = page.add_freetext_annot(
    rect,
    text,
    fill_color=gold,  # fill color
    opacity=1,  # non-transparent
    rotate=0,  # no rotation
    border_width=1,  # border and callout line width
    dashes=None,  # no dashing
    richtext=True,  # this is rich text
    style=ds,  # my styling default
    callout=(p3, p2, rect.tr),  # define end, knee, start points
    line_end=pymupdf.PDF_ANNOT_LE_OPEN_ARROW,  # symbol shown at p3
    border_color=green,
)

doc.save(__file__.replace(".py", ".pdf"), pretty=True)
```

The result looks like this:

19.3 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:

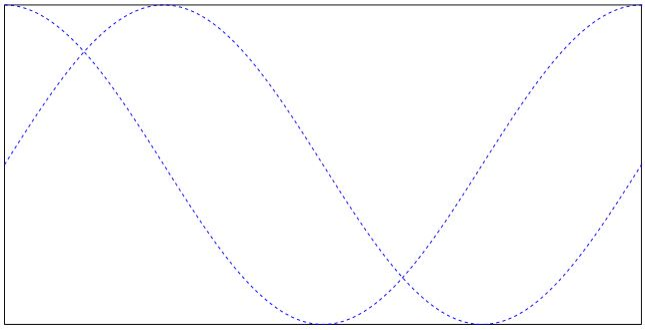
```
import math
import pymupdf

#---------------------------------------------------------------------------
# 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 = pymupdf.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 = pymupdf.open()  # make new PDF
page = doc.new_page()  # give it a page

#---------------------------------------------------------------------------
# add the Ink annotation, consisting of 2 curve segments
#---------------------------------------------------------------------------
annot = page.add_ink_annot((sin_points, cos_points))
# let it look a little nicer
```

(continues on next page)

(continued from previous page)

```
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:



This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTY

DRAWING AND GRAPHICS

Note

When the terms "Drawings" or "Graphics" are mentioned here we are referring to "Vector Graphics" or "Line Art".

Therefore please consider these terms as being synonymous!

PDF files support elementary drawing operations as part of their syntax. These are **vector graphics** and include 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.

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 = pymupdf.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. 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.

"""
```

(continues on next page)

(continued from previous page)

```
import pymupdf
import shapes_and_symbols as sas

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

r = pymupdf.Rect(50, 50, 100, 100)  # first rect to contain a symbol
d = pymupdf.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 = pymupdf.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:

20.1 How to Extract Drawings

- New in v1.18.0

Drawing commands (**vector graphics**) issued by a page can be extracted as a list of dictionaries. 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),
```

(continues on next page)

(continued from previous page)

```
            Point(177.61399841308594, 200.0),
            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 pymupdf
doc = pymupdf.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 = pymupdf.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
```

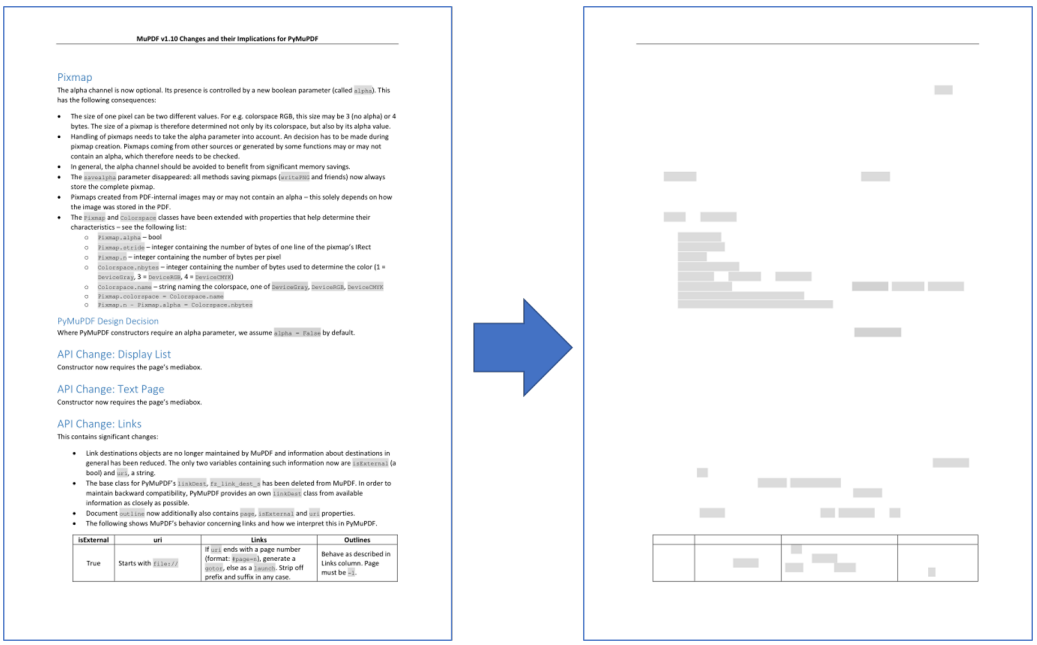
(continues on next page)

(continued from previous page)

```
            shape.draw_quad(item[1])
        elif item[0] == "c":  # curve
            shape.draw_bezier(item[1], item[2], item[3], item[4])
        else:
            raise ValueError("unhandled drawing", item)
    # ------------------------------------------------------
    # all items are drawn, now apply the common properties
    # to finish the path
    # ------------------------------------------------------
    shape.finish(
        fill=path["fill"],  # fill color
        color=path["color"],  # line color
        dashes=path["dashes"],  # line dashing
        even_odd=path.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:



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.

20.2 How to Delete Drawings

To delete drawings/vector graphics we must use a *Redaction Annotation* with the bounding box of the drawing and then **add and apply** a redaction to it to delete it.

The following code shows an example of deleting the first drawing found on the page:

```
paths = page.get_drawings()
rect = paths[0]["rect"]  # rectangle of the 1st drawing
page.add_redact_annot(rect)
page.apply_redactions(0,2,1)  # potentially set options for any of images, drawings, text
```

Note

See *Page.apply_redactions()* for the parameter options which can be sent - you are able to apply deletion options to image, drawing and text objects which are bound by the annotation area.

20.3 How to Draw Graphics

Drawing graphics is as simple as calling the type of *Drawing Method* you may want. You can draw graphics directly on pages or within shape objects.

For example, to draw a circle:

```
# Draw a circle on the page using the Page method
page.draw_circle((center_x, center_y), radius, color=(1, 0, 0), width=2)

# Draw a circle on the page using a Shape object
shape = page.new_shape()
shape.draw_circle((center_x, center_y), radius)
shape.finish(color=(1, 0, 0), width=2)
shape.commit(overlay=True)
```

The *Shape* object can be used to combine multiple drawings that should receive common properties as specified by *Shape.finish()*.

This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTYONE

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.

21.1 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 source[1], that may come from anywhere.
2. Create using the Python API.

Variant using an existing HTML source[1] – which in this case is defined as a constant in the script:

```
import pymupdf

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

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

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

(continues on next page)

[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 - see *CSS Support* for details.

(continued from previous page)

```
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
```

Note

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

```
import pymupdf

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

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

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

The Python API variant – everything is created programmatically:

```
import pymupdf

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

story = pymupdf.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 = pymupdf.DocumentWriter("output.pdf")

more = 1

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

(continues on next page)

(continued from previous page)

```
    story.draw(device)
    writer.end_page()

writer.close()
```

Both variants will produce the same output PDF.

21.2 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 pymupdf

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

# create story, let it look at script folder for resources
story = pymupdf.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 = pymupdf.DocumentWriter("output.pdf")

more = 1

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

writer.close()
```

21.3 How to Read 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 pymupdf

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

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

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

21.4 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).
```

(continues on next page)

(continued from previous page)

```
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.

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 pymupdf

# ---------------------------------------------------------------------------
# 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>'
    "<ol>"
    '<li id="filmtemplate">'
    '<b id="filmtitle"></b>'
    "<dl>"
    '<dt>Director<dd id="director">'
    '<dt>Release Year<dd id="filmyear">'
    '<dt>Cast<dd id="cast">'
    "</dl>"
    "</li>"
    "</ol>"
    "</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
```

(continues on next page)

(continued from previous page)

```
# select statement for the films - let SQL also sort it for us
select_films = """SELECT title, director, year FROM films ORDER BY title"""

# select stament for actors, a skeleton: sub-select by film title
select_casts = """SELECT name FROM actors WHERE film = ? ORDER BY name"""
# ---------------------------------------------------------------------
# define the HTML Story and fill it with database data
# ---------------------------------------------------------------------
story = pymupdf.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 = pymupdf.DocumentWriter(__file__.replace(".py", ".pdf"), "compress")
mediabox = pymupdf.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
```

21.5 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 pymupdf.Story output can be
placed in a rectangle of an existing (!) PDF page.

"""
import io
import os

import pymupdf


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: pymupdf.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 = pymupdf.Rect(0, 0, rect.width, rect.height)
    # this matrix converts mediabox back to input rect
    matrix = mediabox.torect(rect)

    story = pymupdf.Story(text, archive=archive)
    body = story.body
    body.set_properties(font=font)
    writer = pymupdf.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.</p>"""

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

WHERE = pymupdf.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 = pymupdf.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")
```

21.6 How to Make Multi-Columned 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 pymupdf


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

if pymupdf.fitz_fontdescriptors:
    # we want to use the Ubuntu fonts for sans-serif and for monospace
    CSS = pymupdf.css_for_pymupdf_font("ubuntu", archive=arch, name="sans-serif")
    CSS = pymupdf.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 = pymupdf.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 = pymupdf.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 = pymupdf.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 = pymupdf.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 = pymupdf.open("pdf", fileobject)
doc.ez_save(docname)
```

21.7 How to 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
```

(continues on next page)

(continued from previous page)

```
    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.

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 pymupdf


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.
```

(continues on next page)

(continued from previous page)

```
    Notes:
        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 = pymupdf.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)
        [pymupdf.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 = [pymupdf.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()

# ---------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Make the Story object
# ---------------------------------------------------------------
story = pymupdf.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 = pymupdf.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 = pymupdf.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
```

(continues on next page)

(continued from previous page)

```
# Re-open writer output, read its pages and overlay target pages with them.
# The generated pages have same dimension as their targets.
src = pymupdf.open("pdf", fileobject)
doc = pymupdf.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)
```

21.8 How to Output an HTML Table

Outputting HTML tables is supported as follows:

- Flat table layouts are supported ("rows x columns"), no support of the "colspan" / "rowspan" attributes.
- Table header tag *th* supports attribute "scope" with values "row" or "col". Applicable text will be bold by default.
- Column widths are computed automatically based on column content. They cannot be directly set.
- Table **cells may contain images** which will be considered in the column width calculation magic.
- Row heights are computed automatically based on row content - leading to multi-line rows where needed.
- The potentially multiple lines of a table row will always be kept together on one page (respectively "where" rectangle) and not be split.
- Table header rows are only **shown on the first page / "where" rectangle.**
- The "style" attribute is ignored when given directly in HTML table elements. Styling for a table and its elements must happen separately, in CSS source or within the *style* tag.
- Styling for *tr* elements is not supported and ignored. Therefore, a table-wide grid or alternating row background colors are not supported. One of the following example scripts however shows an easy way to deal with this limitation.

Files:

- `docs/samples/table01.py` This script reflects basic features.

```
"""
Demo script for basic HTML table support in Story objects

Outputs a table with three columns that fits on one Letter page.
The content of each row is filled via the Story's template mechanism.
Column widths and row heights are automatically computed by MuPDF.
Some styling via a CSS source is also demonstrated:

- The table header row has a gray background
- Each cell shows a border at its top
- The Story's body uses the sans-serif font family
- The text of one of the columns is set to blue

Dependencies
-------------
PyMuPDF v1.22.0 or later
"""
import pymupdf

table_text = (  # the content of each table row
    (
        "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.""",
    ),
    (
        "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.""",
    ),
    (
        "FDecodeParms",
        "dictionary or array",
        """(Optional; PDF 1.2) A parameter dictionary, or an array of such dictionaries,␣
↪used by the filters specified by FFilter. The same rules apply as for DecodeParms.""",
    ),
    (
        "DecodeParms",
        "dictionary or array",
        """(Optional) A parameter dictionary or an array of such dictionaries, used by␣
```

(continues on next page)

(continued from previous page)

```
    "the filters specified by Filter. If there is only one filter and that filter has␣
    ↪parameters, DecodeParms must be set to the filter's parameter dictionary unless all the␣
    ↪filter's parameters have their default values, in which case the DecodeParms entry may␣
    ↪be omitted. If there are multiple filters and any of the filters has parameters set to␣
    ↪nondefault values, DecodeParms must be an array with one entry for each filter: either␣
    ↪the parameter dictionary for that filter, or the null object if that filter has no␣
    ↪parameters (or if all of its parameters have their default values). If none of the␣
    ↪filters have parameters, or if all their parameters have default values, the␣
    ↪DecodeParms entry may be omitted. (See implementation note 7 in Appendix H.)""",
    ),
    (
        "DL",
        "integer",
        """(Optional; PDF 1.5) A non-negative integer representing the number of bytes␣
    ↪in the decoded (defiltered) stream. It can be used to determine, for example, whether␣
    ↪enough disk space is available to write a stream to a file.\nThis value should be␣
    ↪considered a hint only; for some stream filters, it may not be possible to determine␣
    ↪this value precisely.""",
    ),
    (
        "F",
        "file specification",
        """(Optional; PDF 1.2) The file containing the stream data. If this entry is␣
    ↪present, the bytes between stream and endstream are ignored, the filters are specified␣
    ↪by FFilter rather than Filter, and the filter parameters are specified by FDecodeParms␣
    ↪rather than DecodeParms. However, the Length entry should still specify the number of␣
    ↪those bytes. (Usually, there are no bytes and Length is 0.) (See implementation note␣
    ↪46 in Appendix H.)""",
    ),
)

# Only a minimal HTML source is required to provide the Story's working
HTML = """
<html>
<body><h2>TABLE 3.4 Entries common to all stream dictionaries</h2>
<table>
    <tr>
        <th>KEY</th><th>TYPE</th><th>VALUE</th>
    </tr>
    <tr id="row">
        <td id="col0"></td><td id="col1"></td><td id="col2"></td>
    </tr>
"""

"""
-----------------------------------------------------------------------
Just for demo purposes, set:
- header cell background to gray
- text color in col1 to blue
- a border line at the top of all table cells
- all text to the sans-serif font
-----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
"""
CSS = """th {
    background-color: #aaa;
}

td[id="col1"] {
    color: blue;
}

td, tr {
    border: 1px solid black;
    border-right-width: 0px;
    border-left-width: 0px;
    border-bottom-width: 0px;
}
body {
    font-family: sans-serif;
}
"""

story = pymupdf.Story(HTML, user_css=CSS)  # define the Story
body = story.body  # access the HTML <body> of it
template = body.find(None, "id", "row")  # find the template with name "row"
parent = template.parent  # access its parent i.e., the <table>

for col0, col1, col2 in table_text:
    row = template.clone()  # make a clone of the row template
    # add text to each cell in the duplicated row
    row.find(None, "id", "col0").add_text(col0)
    row.find(None, "id", "col1").add_text(col1)
    row.find(None, "id", "col2").add_text(col2)
    parent.append_child(row)  # add new row to <table>
template.remove()  # remove the template

# Story is ready - output it via a writer
writer = pymupdf.DocumentWriter(__file__.replace(".py", ".pdf"), "compress")
mediabox = pymupdf.paper_rect("letter")  # size of one output page
where = mediabox + (36, 36, -36, -36)  # use this sub-area for the content

more = True  # detects end of output
while more:
    dev = writer.begin_page(mediabox)  # start a page, returning a device
    more, filled = story.place(where)  # compute content fitting into "where"
    story.draw(dev)  # output it to the page
    writer.end_page()  # finalize the page
writer.close()  # close the output
```

- `docs/samples/national-capitals.py` Advanced script extending table output options using simple additional code:
  - Multi-page output simulating **repeating header rows**

- Alternating table row background colors
- Table rows and columns delimited by gridlines
- Table rows dynamically generated / filled with data from an SQL database

```
"""
Demo script using (Py-) MuPDF "Story" feature.

The following features are implemented:

* Use of Story "template" feature to provide row content
* Use database access (SQLITE) to fetch row content
* Use ElementPosition feature to locate cell positions on page
* Simulate feature "Table Header Repeat"
* Simulate feature "Cell Grid Lines"

"""
import io
import sqlite3
import sys

import pymupdf

"""
Table data. Used to populate a temporary SQL database, which will be processed by the␣
↪script.
Its only purpose is to avoid carrying around a separate database file.
"""
# codespell:ignore-begin
table_data = """China;Beijing;21542000;1.5%;2018
Japan;Tokyo;13921000;11.2%;2019
DR Congo;Kinshasa;12691000;13.2%;2017
Russia;Moscow;12655050;8.7%;2021
Indonesia;Jakarta;10562088;3.9%;2020
Egypt;Cairo;10107125;9.3%;2022
South Korea;Seoul;9508451;18.3%;2022
Mexico;Mexico City;9209944;7.3%;2020
United Kingdom;London;9002488;13.4%;2020
Bangladesh;Dhaka;8906039;5.3%;2011
Peru;Lima;8852000;26.3%;2012
Iran;Tehran;8693706;9.9%;2016
Thailand;Bangkok;8305218;11.6%;2010
Vietnam;Hanoi;8053663;8.3%;2019
Iraq;Baghdad;7682136;17.6%;2021
Saudi Arabia;Riyadh;7676654;21.4%;2018
Hong Kong;Hong Kong;7291600;100%;2022
Colombia;Bogotá;7181469;13.9%;2011
Chile;Santiago;6310000;32.4%;2012
Turkey;Ankara;5747325;6.8%;2021
Singapore;Singapore;5453600;91.8%;2021
Afghanistan;Kabul;4601789;11.5%;2021
Kenya;Nairobi;4397073;8.3%;2019
```

(continues on next page)

(continued from previous page)

```
Jordan;Amman;4061150;36.4%;2021
Algeria;Algiers;3915811;8.9%;2011
Germany;Berlin;3677472;4.4%;2021
Spain;Madrid;3305408;7.0%;2021
Ethiopia;Addis Ababa;3040740;2.5%;2012
Kuwait;Kuwait City;2989000;70.3%;2018
Guatemala;Guatemala City;2934841;16.7%;2020
South Africa;Pretoria;2921488;4.9%;2011
Ukraine;Kyiv;2920873;6.7%;2021
Argentina;Buenos Aires;2891082;6.4%;2010
North Korea;Pyongyang;2870000;11.1%;2016
Uzbekistan;Tashkent;2860600;8.4%;2022
Italy;Rome;2761632;4.7%;2022
Ecuador;Quito;2800388;15.7%;2020
Cameroon;Yaoundé;2765568;10.2%;2015
Zambia;Lusaka;2731696;14.0%;2020
Sudan;Khartoum;2682431;5.9%;2012
Brazil;Brasília;2648532;1.2%;2012
Taiwan;Taipei (de facto);2608332;10.9%;2020
Yemen;Sanaa;2575347;7.8%;2012
Angola;Luanda;2571861;7.5%;2020
Burkina Faso;Ouagadougou;2453496;11.1%;2019
Ghana;Accra;2388000;7.3%;2017
Somalia;Mogadishu;2388000;14.0%;2021
Azerbaijan;Baku;2303100;22.3%;2022
Cambodia;Phnom Penh;2281951;13.8%;2019
Venezuela;Caracas;2245744;8.0%;2016
France;Paris;2139907;3.3%;2022
Cuba;Havana;2132183;18.9%;2020
Zimbabwe;Harare;2123132;13.3%;2012
Syria;Damascus;2079000;9.7%;2019
Belarus;Minsk;1996553;20.8%;2022
Austria;Vienna;1962779;22.0%;2022
Poland;Warsaw;1863056;4.9%;2021
Philippines;Manila;1846513;1.6%;2020
Mali;Bamako;1809106;8.3%;2009
Malaysia;Kuala Lumpur;1782500;5.3%;2019
Romania;Bucharest;1716983;8.9%;2021
Hungary;Budapest;1706851;17.6%;2022
Congo;Brazzaville;1696392;29.1%;2015
Serbia;Belgrade;1688667;23.1%;2021
Uganda;Kampala;1680600;3.7%;2019
Guinea;Conakry;1660973;12.3%;2014
Mongolia;Ulaanbaatar;1466125;43.8%;2020
Honduras;Tegucigalpa;1444085;14.0%;2021
Senegal;Dakar;1438725;8.5%;2021
Niger;Niamey;1334984;5.3%;2020
Uruguay;Montevideo;1319108;38.5%;2011
Bulgaria;Sofia;1307439;19.0%;2021
Oman;Muscat;1294101;28.6%;2021
Czech Republic;Prague;1275406;12.1%;2022
Madagascar;Antananarivo;1275207;4.4%;2018
```

(continues on next page)

(continued from previous page)

```
Kazakhstan;Astana;1239900;6.5%;2022
Nigeria;Abuja;1235880;0.6%;2011
Georgia;Tbilisi;1201769;32.0%;2022
Mauritania;Nouakchott;1195600;25.9%;2019
Qatar;Doha;1186023;44.1%;2020
Libya;Tripoli;1170000;17.4%;2019
Myanmar;Naypyidaw;1160242;2.2%;2014
Rwanda;Kigali;1132686;8.4%;2012
Mozambique;Maputo;1124988;3.5%;2020
Dominican Republic;Santo Domingo;1111838;10.0%;2010
Armenia;Yerevan;1096100;39.3%;2021
Kyrgyzstan;Bishkek;1074075;16.5%;2021
Sierra Leone;Freetown;1055964;12.5%;2015
Nicaragua;Managua;1055247;15.4%;2020
Canada;Ottawa;1017449;2.7%;2021
Pakistan;Islamabad;1014825;0.4%;2017
Liberia;Monrovia;1010970;19.5%;2008
United Arab Emirates;Abu Dhabi;1010092;10.8%;2020
Malawi;Lilongwe;989318;5.0%;2018
Haiti;Port-au-Prince;987310;8.6%;2015
Sweden;Stockholm;978770;9.4%;2021
Eritrea;Asmara;963000;26.6%;2020
Israel;Jerusalem;936425;10.5%;2019
Laos;Vientiane;927724;12.5%;2019
Chad;N'Djamena;916000;5.3%;2009
Netherlands;Amsterdam;905234;5.2%;2022
Central African Republic;Bangui;889231;16.3%;2020
Panama;Panama City;880691;20.2%;2013
Tajikistan;Dushanbe;863400;8.9%;2020
Nepal;Kathmandu;845767;2.8%;2021
Togo;Lomé;837437;9.7%;2010
Turkmenistan;Ashgabat;791000;12.5%;2017
Moldova;Chişinău;779300;25.5%;2019
Croatia;Zagreb;769944;19.0%;2021
Gabon;Libreville;703904;30.1%;2013
Norway;Oslo;697010;12.9%;2021
Macau;Macau;671900;97.9%;2022
United States;Washington D.C.;670050;0.2%;2021
Jamaica;Kingston;662491;23.4%;2019
Finland;Helsinki;658864;11.9%;2021
Tunisia;Tunis;638845;5.2%;2014
Denmark;Copenhagen;638117;10.9%;2021
Greece;Athens;637798;6.1%;2021
Latvia;Riga;605802;32.3%;2021
Djibouti;Djibouti (city);604013;54.6%;2012
Ireland;Dublin;588233;11.8%;2022
Morocco;Rabat;577827;1.6%;2014
Lithuania;Vilnius;576195;20.7%;2022
El Salvador;San Salvador;570459;9.0%;2019
Albania;Tirana;557422;19.5%;2011
North Macedonia;Skopje;544086;25.9%;2015
South Sudan;Juba;525953;4.9%;2017
```

(continues on next page)

(continued from previous page)

```
Paraguay;Asunción;521559;7.8%;2020
Portugal;Lisbon;509614;5.0%;2020
Guinea-Bissau;Bissau;492004;23.9%;2015
Slovakia;Bratislava;440948;8.1%;2020
Estonia;Tallinn;438341;33.0%;2021
Australia;Canberra;431380;1.7%;2020
Namibia;Windhoek;431000;17.0%;2020
Tanzania;Dodoma;410956;0.6%;2012
Papua New Guinea;Port Moresby;364145;3.7%;2011
Ivory Coast;Yamoussoukro;361893;1.3%;2020
Lebanon;Beirut;361366;6.5%;2014
Bolivia;Sucre;360544;3.0%;2022
Puerto Rico (US);San Juan;342259;10.5%;2020
Costa Rica;San José;342188;6.6%;2018
Lesotho;Maseru;330760;14.5%;2016
Cyprus;Nicosia;326739;26.3%;2016
Equatorial Guinea;Malabo;297000;18.2%;2018
Slovenia;Ljubljana;285604;13.5%;2021
East Timor;Dili;277279;21.0%;2015
Bosnia and Herzegovina;Sarajevo;275524;8.4%;2013
Bahamas;Nassau;274400;67.3%;2016
Botswana;Gaborone;273602;10.6%;2020
Benin;Porto-Novo;264320;2.0%;2013
Suriname;Paramaribo;240924;39.3%;2012
India;New Delhi;249998;0.0%;2011
Sahrawi Arab Democratic Republic;Laayoune (claimed) - Tifariti (de facto);217732 - 3000;
↪-;2014
New Zealand;Wellington;217000;4.2%;2021
Bahrain;Manama;200000;13.7%;2020
Kosovo;Pristina;198897;12.0%;2011
Montenegro;Podgorica;190488;30.3%;2020
Belgium;Brussels;187686;1.6%;2022
Cape Verde;Praia;159050;27.1%;2017
Mauritius;Port Louis;147066;11.3%;2018
Curaçao (Netherlands);Willemstad;136660;71.8%;2011
Burundi;Gitega;135467;1.1%;2020
Switzerland;Bern (de facto);134591;1.5%;2020
Transnistria;Tiraspol;133807;38.5%;2015
Maldives;Malé;133412;25.6%;2014
Iceland;Reykjavík;133262;36.0%;2021
Luxembourg;Luxembourg City;124509;19.5%;2021
Guyana;Georgetown;118363;14.7%;2012
Bhutan;Thimphu;114551;14.7%;2017
Comoros;Moroni;111326;13.5%;2016
Barbados;Bridgetown;110000;39.1%;2014
Sri Lanka;Sri Jayawardenepura Kotte;107925;0.5%;2012
Brunei;Bandar Seri Begawan;100700;22.6%;2007
Eswatini;Mbabane;94874;8.0%;2010
New Caledonia (France);Nouméa;94285;32.8%;2019
Fiji;Suva;93970;10.2%;2017
Solomon Islands;Honiara;92344;13.0%;2021
Republic of Artsakh;Stepanakert;75000;62.5%;2021
```

(continues on next page)

(continued from previous page)

```
Gambia;Banjul;73000;2.8%;2013
São Tomé and Príncipe;São Tomé;71868;32.2%;2015
Kiribati;Tarawa;70480;54.7%;2020
Vanuatu;Port Vila;51437;16.1%;2016
Northern Mariana Islands (USA);Saipan;47565;96.1%;2017
Samoa;Apia;41611;19.0%;2021
Palestine;Ramallah (de facto);38998;0.8%;2017
Monaco;Monaco;38350;104.5%;2020
Jersey (UK);Saint Helier;37540;34.2%;2018
Trinidad and Tobago;Port of Spain;37074;2.4%;2011
Cayman Islands (UK);George Town;34399;50.5%;2021
Gibraltar (UK);Gibraltar;34003;104.1%;2020
Grenada;St. George's;33734;27.1%;2012
Aruba (Netherlands);Oranjestad;28294;26.6%;2010
Isle of Man (UK);Douglas;27938;33.2%;2011
Marshall Islands;Majuro;27797;66.1%;2011
Tonga;Nukualofa;27600;26.0%;2022
Seychelles;Victoria;26450;24.8%;2010
French Polynesia (France);Papeete;26926;8.9%;2017
Andorra;Andorra la Vella;22873;28.9%;2022
Faroe Islands (Denmark);Tórshavn;22738;43.0%;2022
Antigua and Barbuda;St. John's;22219;23.8%;2011
Belize;Belmopan;20621;5.2%;2016
Saint Lucia;Castries;20000;11.1%;2013
Guernsey (UK);Saint Peter Port;18958;30.1%;2019
Greenland (Denmark);Nuuk;18800;33.4%;2021
Dominica;Roseau;14725;20.3%;2011
Saint Kitts and Nevis;Basseterre;14000;29.4%;2018
Saint Vincent and the Grenadines;Kingstown;12909;12.4%;2012
British Virgin Islands (UK);Road Town;12603;40.5%;2012
Åland (Finland);Mariehamn;11736;39.0%;2021
U.S. Virgin Islands (US);Charlotte Amalie;14477;14.5%;2020
Micronesia;Palikir;6647;5.9%;2010
Tuvalu;Funafuti;6320;56.4%;2017
Malta;Valletta;5827;1.1%;2019
Liechtenstein;Vaduz;5774;14.8%;2021
Saint Pierre and Miquelon (France);Saint-Pierre;5394;91.7%;2019
Cook Islands (NZ);Avarua;4906;28.9%;2016
San Marino;City of San Marino;4061;12.0%;2021
Turks and Caicos Islands (UK);Cockburn Town;3720;8.2%;2016
American Samoa (USA);Pago Pago;3656;8.1%;2010
Saint Martin (France);Marigot;3229;10.1%;2017
Saint Barthélemy (France);Gustavia;2615;24.1%;2010
Falkland Islands (UK);Stanley;2460;65.4%;2016
Svalbard (Norway);Longyearbyen;2417;82.2%;2020
Sint Maarten (Netherlands);Philipsburg;1894;4.3%;2011
Christmas Island (Australia);Flying Fish Cove;1599;86.8%;2016
Anguilla (UK);The Valley;1067;6.8%;2011
Guam (US);Hagåtña;1051;0.6%;2010
Wallis and Futuna (France);Mata Utu;1029;8.9%;2018
Bermuda (UK);Hamilton;854;1.3%;2016
Nauru;Yaren (de facto);747;6.0%;2011
```

(continues on next page)

(continued from previous page)

```
Saint Helena (UK);Jamestown;629;11.6%;2016
Niue (NZ);Alofi;597;30.8%;2017
Tokelau (NZ);Atafu;541;29.3%;2016
Vatican City;Vatican City (city-state);453;100%;2019
Montserrat (UK);Brades (de facto) - Plymouth (de jure);449 - 0;-;2011
Norfolk Island (Australia);Kingston;341;-;2015
Palau;Ngerulmud;271;1.5%;2010
Cocos (Keeling) Islands (Australia);West Island;134;24.6%;2011
Pitcairn Islands (UK);Adamstown;40;100.0%;2021
South Georgia and the South Sandwich Islands (UK);King Edward Point;22;73.3%;2018"""
# codespell:ignore-end

# ---------------------------------------------------------------------
# HTML template for the report. We define no table header <th> items
# because this is done in post processing.
# The actual template part is the table row, identified by id "row".
# The content of each cell will be filled using the respective id.
# ---------------------------------------------------------------------
HTML = """
    <h1 style="text-align:center">World Capital Cities</h1>
    <p><i>Percent "%" is city population as a percentage of the country, as of "Year".</
↪i>
    </p><p></p>
    <table>
    <tr id="row">
        <td id="country"></td>
        <td id="capital"></td>
        <td id="population"></td>
        <td id="percent"></td>
        <td id="year"></td>
    </tr>
    </table>
"""

# ---------------------------------------------------------------------
# Sets font-family globally to sans-serif, and text-align to right
# for the numerical table columns.
# ---------------------------------------------------------------------
CSS = """
body {
    font-family: sans-serif;
}
td[id="population"], td[id="percent"], td[id="year"] {
    text-align: right;
    padding-right: 2px;
}"""

# ---------------------------------------------------------------------
# recorder function for cell positions
# ---------------------------------------------------------------------
coords = {}  # stores cell gridline coordinates
```

(continues on next page)

(continued from previous page)

```
def recorder(elpos):
    """We only record positions of table rows and cells.

    Information is stored in "coords" with page number as key.
    """
    global coords  # dictionary of row and cell coordinates per page
    if elpos.open_close != 2:  # only consider coordinates provided at "close"
        return
    if elpos.id not in ("row", "country", "capital", "population", "percent", "year"):
        return  # only look at row / cell content

    rect = pymupdf.Rect(elpos.rect)  # cell rectangle
    if rect.y1 > elpos.filled:  # ignore stuff below the filled rectangle
        return

    # per page, we store the floats top-most y, right-most x, column left
    # and row bottom borders.
    x, y, x1, y0 = coords.get(elpos.page, (set(), set(), 0, sys.maxsize))

    if elpos.id != "row":
        x.add(rect.x0)  # add cell left border coordinate
        if rect.x1 > x1:  # store right-most cell border on page
            x1 = rect.x1
    else:
        y.add(rect.y1)  # add row bottom border coordinate
        if rect.y0 < y0:  # store top-most cell border per page
            y0 = rect.y0

    coords[elpos.page] = (x, y, x1, y0)  # write back info per page
    return


# ---------------------------------------------------------------------
# define database access: make an intermediate memory database for
# our demo purposes.
# ---------------------------------------------------------------------
dbfilename = ":memory:"  # the SQLITE database file name
database = sqlite3.connect(dbfilename)  # open database
cursor = database.cursor()  # multi-purpose database cursor

# Define and fill the SQLITE database
cursor.execute(
    """CREATE TABLE capitals (Country text, Capital text, Population text, Percent text,␣
↪Year text)"""
)

for value in table_data.splitlines():
    cursor.execute("INSERT INTO capitals VALUES (?,?,?,?,?)", value.split(";"))

# select statement for the rows - let SQL also sort it for us
select = """SELECT * FROM capitals ORDER BY "Country" """
```

(continues on next page)

(continued from previous page)

```
# ---------------------------------------------------------------------
# define the HTML Story and fill it with database data
# ---------------------------------------------------------------------
story = pymupdf.Story(HTML, user_css=CSS)
body = story.body  # access the HTML body detail

template = body.find(None, "id", "row")  # find the template part
table = body.find("table", None, None)  # find start of table

# read the rows 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.execute(select)  # execute cursor, and ...
rows = cursor.fetchall()  # read out what was found
database.close()  # no longer needed

for country, capital, population, percent, year in rows:  # iterate through the row
    row = template.clone()  # clone the template to report each row
    row.find(None, "id", "country").add_text(country)
    row.find(None, "id", "capital").add_text(capital)
    row.find(None, "id", "population").add_text(population)
    row.find(None, "id", "percent").add_text(percent)
    row.find(None, "id", "year").add_text(year)

    table.append_child(row)

template.remove()  # remove the template

# ---------------------------------------------------------------------
# generate the PDF and write it to memory
# ---------------------------------------------------------------------
fp = io.BytesIO()
writer = pymupdf.DocumentWriter(fp)
mediabox = pymupdf.paper_rect("letter")  # use pages in Letter format
where = mediabox + (36, 36, -36, -72)  # leave page borders
more = True
page = 0
while more:
    dev = writer.begin_page(mediabox)  # make a new page
    if page > 0:  # leave room above the cells for inserting header row
        delta = (0, 20, 0, 0)
    else:
        delta = (0, 0, 0, 0)
    more, filled = story.place(where + delta)  # arrange content on this rectangle
    story.element_positions(recorder, {"page": page, "filled": where.y1})
    story.draw(dev)  # write content to page
    writer.end_page()  # finish the page
    page += 1
writer.close()  # close the PDF

# ---------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# re-open memory PDF for inserting gridlines and header rows
# ----------------------------------------------------------------------
doc = pymupdf.open("pdf", fp)
for page in doc:
    page.wrap_contents()  # ensure all "cm" commands are properly wrapped
    x, y, x1, y0 = coords[page.number]  # read coordinates of the page
    x = sorted(list(x)) + [x1]  # list of cell left-right borders
    y = [y0] + sorted(list(y))  # list of cell top-bottom borders
    shape = page.new_shape()  # make a canvas to draw upon

    for item in y:  # draw horizontal lines (one under each row)
        shape.draw_line((x[0] - 2, item), (x[-1] + 2, item))

    for i in range(len(y)):  # alternating row coloring
        if i % 2:
            rect = (x[0] - 2, y[i - 1], x[-1] + 2, y[i])
            shape.draw_rect(rect)

    for i in range(len(x)):  # draw vertical lines
        d = 2 if i == len(x) - 1 else -2
        shape.draw_line((x[i] + d, y[0]), (x[i] + d, y[-1]))

    # Write header row above table content
    y0 -= 5  # bottom coord for header row text
    shape.insert_text((x[0], y0), "Country", fontname="hebo", fontsize=12)
    shape.insert_text((x[1], y0), "Capital", fontname="hebo", fontsize=12)
    shape.insert_text((x[2], y0), "Population", fontname="hebo", fontsize=12)
    shape.insert_text((x[3], y0), "  %", fontname="hebo", fontsize=12)
    shape.insert_text((x[4], y0), "Year", fontname="hebo", fontsize=12)

    # Write page footer
    y0 = page.rect.height - 50  # top coordinate of footer bbox
    bbox = pymupdf.Rect(0, y0, page.rect.width, y0 + 20)  # footer bbox
    page.insert_textbox(
        bbox,
        f"World Capital Cities, Page {page.number+1} of {doc.page_count}",
        align=pymupdf.TEXT_ALIGN_CENTER,
    )
    shape.finish(width=0.3, color=0.5, fill=0.9)  # rectangles and gray lines
    shape.commit(overlay=False)  # put the drawings in background

doc.subset_fonts()
doc.save(__file__.replace(".py", ".pdf"), deflate=True, garbage=4, pretty=True)
doc.close()
```

21.9 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 pymupdf

MEDIABOX = pymupdf.paper_rect("letter")  # output page format: Letter
GRIDSPACE = pymupdf.Rect(100, 100, 400, 400)
GRID = pymupdf.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 = pymupdf.DocumentWriter(__file__.replace(".py", ".pdf"))  # create the writer

device = writer.begin_page(MEDIABOX)  # make new page
for i, text in enumerate(text_table):
    story = pymupdf.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
```

21.10 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
--------------------------------
```

(continues on next page)

(continued from previous page)

```
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 pymupdf.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 Stoy callback feature
   - uses Story also for making the TOC page(s)

"""
import io
import os
import time

import pymupdf

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 = pymupdf.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
        TOC.append(
            (
                level,
                elpos.text,
                elpos.page + 1,
                y0,
            )
        )
        return

    CURRENT_ID = elpos.id if elpos.id else ""  # update for footer line
```

(continues on next page)

(continued from previous page)

```
    return


def header_story(text):
    """Make the page header"""
    header = pymupdf.Story()
    hdr_body = header.body
    hdr_body.add_paragraph().set_properties(
        align=pymupdf.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 = pymupdf.Story()
    ftr_body = footer.body
    ftr_body.add_paragraph().set_properties(
        bgcolor="#eee",
        align=pymupdf.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 = pymupdf.DocumentWriter(outfile, "")
    print_time = time.strftime("%Y-%m-%d %H:%M:%S (%z)")
    thispath = os.path.abspath(os.curdir)
    basename = os.path.basename(thispath)

    story = pymupdf.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:
```

(continues on next page)

(continued from previous page)

```
        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(pymupdf.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
        # 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:
```

(continues on next page)

(continued from previous page)

```
            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 = pymupdf.open("pdf", fileptr1)
    old_count = doc.page_count
    # --------------------------------------------------------------------------
    # Post-processing step to make / insert the toc
    # This also works using pymupdf.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 = pymupdf.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]})"
        )
    fileptr2 = io.BytesIO()  # put TOC pages to a separate PDF initially
    writer = pymupdf.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)
```

(continues on next page)

(continued from previous page)

```
        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 = pymupdf.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=pymupdf.TEXTFLAGS_SEARCH)
            if rl != []:  # this text must be on next page
                break
        else:
            assert 0, f'Cannot find {item[1]=} in {len(pages)=}.'
        rect = rl[0]  # rectangle of TOC item text
        link = {  # make a link from it
            "kind": pymupdf.LINK_GOTO,
            "from": rect,
            "to": pymupdf.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.

21.11 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 pymupdf
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"
    },
    {
        "name" :            "Golden Hall",
        "temple" :          "Murō-ji",
        "founded" :         "early Heian period",
        "region" :          "Uda, Nara",
        "position" :        "34.536586819357986,136.0395548452301"
    },
    {
```

(continues on next page)

(continued from previous page)

```
        "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",
        "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",
```

(continues on next page)

(continued from previous page)


```
        "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",
        "position" :        "36.735689,137.010019"
    },
    {
        "name" :            "Lecture Hall",
        "temple" :          "Tōshōdai-ji",
        "founded" :         "Nara period, 763",
        "region" :          "Nara, Nara",
        "position" :        "34.675933,135.784842"
    },
```


(continues on next page)

(continued from previous page)

```
    {
        "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 = pymupdf.paper_rect("letter")  # output page format: Letter
WHERE = MEDIABOX + (36, 36, -36, -36)
writer = pymupdf.DocumentWriter("json-example.pdf")  # create the writer

story = pymupdf.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()
            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
```

(continues on next page)

(continued from previous page)

```
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
```

21.12 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.

21.12.1 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 pymupdf


# 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>
```

(continues on next page)

(continued from previous page)

```
<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 = pymupdf.Rect(0, 0, page_w, page_h)
    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 = pymupdf.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 = pymupdf.Story(html, em=8)
```

(continues on next page)

(continued from previous page)

```
out_path = __file__.replace('.py', '.pdf')
writer = pymupdf.DocumentWriter(out_path)

story.write(writer, rectfn)
writer.close()
```

21.12.2 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 `pymupdf.Story.write_stabilized()`.

`pymupdf.Story.write_stabilized()` is similar to `pymupdf.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
`pymupdf.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 pymupdf


def rectfn(rect_num, filled):
    '''
    We return one rect per page.
    '''
    rect = pymupdf.Rect(10, 20, 290, 380)
    mediabox = pymupdf.Rect(0, 0, 300, 400)
    #print(f'rectfn(): rect_num={rect_num} filled={filled}')
    return mediabox, rect, None


def contentfn(positions):
```

(continues on next page)

(continued from previous page)

```
    '''
    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.
            <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:
```

(continues on next page)

(continued from previous page)

```
        f.write(ret)
    return ret;


out_path = __file__.replace('.py', '.pdf')
writer = pymupdf.DocumentWriter(out_path)
pymupdf.Story.write_stabilized(writer, contentfn, rectfn)
writer.close()
```

21.12.3 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 `pymupdf.Story.write_stabilized_with_links()`.

`pymupdf.Story.write_stabilized_links()` is similar to
`pymupdf.Story.write_stabilized()` except that it creates a PDF `pymupdf.Document`
that contains PDF links generated from all internal links in the original html.
"""

import textwrap

import pymupdf


def rectfn(rect_num, filled):
    '''
    We return one rect per page.
    '''
    rect = pymupdf.Rect(10, 20, 290, 380)
    mediabox = pymupdf.Rect(0, 0, 300, 400)
    #print(f'rectfn(): rect_num={rect_num} filled={filled}')
    return mediabox, rect, None
```

(continues on next page)

(continued from previous page)

```
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.
            <ul>
            <li>External <a href="https://artifex.com/">link to https://artifex.com/</a>.
            <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.
```

(continues on next page)

(continued from previous page)

```
            <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 = pymupdf.Story.write_stabilized_with_links(contentfn, rectfn)
document.save(out_path)
```

This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTYTWO

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 "undone" resp. "redone".
- 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.

22.1 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 pymupdf
>>> doc=pymupdf.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")
>>> 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())
```

(continues on next page)

(continued from previous page)

```
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!
```

22.2 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=pymupdf.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(f"insert-{i}")
        page.insert_text((100, 100 + 20*i), f"text line {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)
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTYTHREE

MULTIPROCESSING

does not support running on multiple threads - doing so may cause incorrect behaviour or even crash Python itself.

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 pymupdf

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


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

    Notes:
        The PyMuPDF document cannot be part of the argument, because that
        cannot be pickled. So we are being passed in just its filename.
        This is no performance issue, because we are a separate process and
        need to open the document anyway.
        Any page-specific function can be processed here - rendering is just
        an example - text extraction might be another.
```

(continues on next page)

(continued from previous page)

```
        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 = pymupdf.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(f"Processed page numbers {seg_from} through {seg_to - 1}")


if __name__ == "__main__":
    t0 = mytime()  # start a timer
    filename = sys.argv[1]
    mat = pymupdf.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(f"Starting {cpu} processes for '{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(f"Total time {round(t1 - t0, 2):g} seconds")
```

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

```
"""
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 pymupdf

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

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

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

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

    def onTimerGetPage(self):
        try:
            ret = self.queDoc.get(False)
            if isinstance(ret, int):
                self.timerWaiting.stop()
                self.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 = pymupdf.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())
```

This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTYFOUR

OPTIONAL CONTENT SUPPORT

This document explains PyMuPDF's support of the PDF concept **"Optional Content"**.

24.1 Introduction: The Optional Content Concept

Optional Content in PDF is a way to show or hide parts of a document based on certain conditions: Parameters that can be set to ON or to OFF when using a supporting PDF consumer (viewer), or programmatically.

This capability is useful in items such as CAD drawings, layered artwork, maps, and multi-language documents. Typical uses include showing or hiding details of complex vector graphics like geographical maps, technical devices, architectural designs and similar, including automatically switching between different zooming levels. Other use cases may be to automatically show different detail levels when displaying a document on screen as opposed to printing it.

Special PDF objects, so-called **Optional Content Groups** (OCGs) are used to define these different *layers* of content.

Assigning an OCG to a "normal" PDF object (like a text or an image) causes that object to be visible or hidden, depending on the current state of the assigned OCG.

To ease definition of the overall configuration of a PDF's Optional Content, OCGs can be organized in higher level groupings, called **OC Configurations**. Each configuration being a collection of OCGs, together with each OCG's desired initial visibility state. Selecting one of these configurations (via the PDF viewer or programmatically) causes a corresponding visibility change of all affected PDF objects throughout the document.

Except for the default one, OC Configurations are optional.

For more explanations and additional background please refer to PDF specification manuals.

24.2 PyMuPDF Support for PDF Optional Content

PyMuPDF offers full support for viewing, defining, changing and deleting Option Content Groups, Configurations, maintaining the assignment of OCGs to PDF objects and programmatically switching between OC Configurations and the visibility states of each single OCG.

24.3 How to Add Optional Content

This is as simple as adding an Optional Content Group, OCG, to a PDF: *Document.add_ocg()*.

If previously the PDF had no OC support at all, the required setup (like defining the default OC Configuration) will be done at this point automatically.

The method returns an *xref* of the created OCG. Use this xref to associate (mark) any PDF object with it, that you want to make dependent on this OCG's state. For example, you can insert an image on a page and refer to the xref like this:

```
img_xref = page.insert_image(rect, filename="image.file", oc=xref)
```

If you want to put an **existing** image under the control of an OCG, you must first find out the image's xref number (called `img_xref` here) and then do `doc.set_oc(img_xref, xref)`. After this, the image will be (in-) visible everywhere throughout the document if the OCG's state is "ON", respectively "OFF". You can also assign a different OCG with this method.

To **remove** an OCG from an image, do `doc.set_oc(img_xref, 0)`.

One single OCG can be assigned to multiple PDF objects to control their visibility.

24.4 How to Define Complex Optional Content Conditions

Sophisticated logical conditions can be established to address complex visibility needs.

For example, you might want to create a multi-language document, so the user may switch between languages as required.

Please have a look at this Jupyter Notebook and execute it as desired.

Certainly, your requirements may even be more complex and involve multiple OCGs with ON/OFF states that are connected by some kind of logical relationship – but it should give you an impression of what is possible and how to plan your next steps.

This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTYFIVE

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.

25.1 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(f"object {xref} (stream: {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 ]
```

(continues on next page)

(continued from previous page)

```
    /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.

25.2 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.

25.3 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
>>> 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.

25.4 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 pymupdf
>>> doc=pymupdf.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*.

25.5 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 *xref* section (indicates incremental updates).
Root	dictionary	(indirect) Pointer to the catalog. See previous section.
Encrypt	dictionary	Pointer to encryption object (encrypted files only).
Info	dictionary	(indirect) Pointer to information (metadata).
ID	array	File identifier consisting of two byte strings.
XRefStm	int	Offset of a cross-reference stream. See *Adobe PDF References* 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 pymupdf
>>> doc=pymupdf.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
>>
>>>
```

25.6 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="\ufeff" id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/" x:xmptk="3.1-702">
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
...
omitted data
...
<?xpacket end="w"?>
```

Using some XML package, the XML data can be interpreted and / or modified and then stored back. 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()
```

25.7 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
# ---------------------------------
```

(continues on next page)

(continued from previous page)

```
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:20101024170344O6'-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", pymupdf.get_pdf_str(" is Beijing"))
#
# after executing the previous code snippet, we will see this:
pprint(metadata)
{'Author': 'PRINCE',
 'CreationDate': "D:20101024170344O6'-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.

25.8 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 pymupdf
In [2]: doc = pymupdf.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(f"{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
```

(continues on next page)

(continued from previous page)

```
  /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 en 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
      /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
```

(continues on next page)

(continued from previous page)

```
    >>
  >>
  /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 <<
    /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
```

(continues on next page)

(continued from previous page)

```
--- ... ---
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.

This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTYSIX

COMMON ISSUES AND THEIR SOLUTIONS

26.1 How To Dynamically Clean Up Corrupt *PDFs*

This shows a potential use of 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 pymupdf

#---------------------------------------
# '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 = pymupdf.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.

26.2 How to Convert Any Document to

Here is a script that converts any *supported document* to a . These include XPS, EPUB, FB2, CBZ and 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 pymupdf
if not (list(map(int, pymupdf.VersionBind.split("."))) >= [1,14,0]):
    raise SystemExit("need PyMuPDF v1.14.0+")
fn = sys.argv[1]

print(f"Converting '{fn}' to '{fn}.pdf'")

doc = pymupdf.open(fn)

b = doc.convert_to_pdf()  # convert to pdf
pdf = pymupdf.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" + pymupdf.VersionBind

if not meta["creator"]:
```

(continues on next page)

(continued from previous page)

```
    meta["creator"] = "PyMuPDF PDF converter"
meta["modDate"] = pymupdf.get_pdf_now()
meta["creationDate"] = meta["modDate"]
pdf.set_metadata(meta)

# now process the links
link_cnti = 0
link_skip = 0
for pinput in doc:  # iterate through input pages
    links = pinput.get_links()  # get list of links
    link_cnti += len(links)  # count how many
    pout = pdf[pinput.number]  # read corresp. output page
    for l in links:  # iterate though the links
        if l["kind"] == pymupdf.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_cnti > 0:
    print(f"Skipped {link_skip} named links of a total of {link_cnti} in input.")
```

26.3 Changing Annotations: Unexpected Behaviour

26.3.1 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.

26.3.2 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.

26.3.3 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_oc()* (add / remove reference to optional content information)
- *Annot.set_open()*
- *Annot.update_file()* (file attachment changes)

26.4 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.

26.4.1 Problem: no text is extracted

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

26.4.2 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.

26.4.3 Solution

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

26.4.4 Problem: unreadable text

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

26.4.5 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 haved used a font that displays readable text, but on purpose obfuscates the originating corresponding unicode character.

26.4.6 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.

This documentation covers all versions up to 1.28.2.

This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTYSEVEN

CLASSES

27.1 Annot

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.

Attribute	Short Description
Annot.delete_responses()	delete all responding annotations
Annot.get_file()	get attached file content
Annot.get_oc()	get *xref* of an *OCG* / *OCMD*
Annot.get_pixmap()	image of the annotation as a pixmap
Annot.get_sound()	get the sound of an audio annotation
Annot.get_text()	extract annotation text
Annot.get_textbox()	extract annotation text
Annot.get_textpage()	create a TextPage for the annotation
Annot.set_border()	set annotation's border properties
Annot.set_blendmode()	set annotation's blend mode
Annot.set_colors()	set annotation's colors
Annot.set_flags()	set annotation's flags field
Annot.set_irt_xref()	define the annotation to being "In Response To"
Annot.set_name()	set annotation's name field
Annot.set_oc()	set *xref* to an *OCG* / *OCMD*
Annot.set_opacity()	change transparency
Annot.set_open()	open / close annotation or its Popup
Annot.set_popup()	create a Popup for the annotation
Annot.set_rect()	change annotation rectangle
Annot.set_rotation()	change rotation
Annot.update_file()	update attached file content
Annot.update()	apply accumulated annot changes
Annot.blendmode	annotation BlendMode
Annot.border	border details
Annot.colors	border / background and fill colors
Annot.file_info	get attached file information

continues on next page

Table 1 – continued from previous page

Attribute	Short Description
Annot.flags	annotation flags
Annot.has_popup	whether annotation has a Popup
Annot.irt_xref	annotation to which this one responds
Annot.info	various information
Annot.is_open	whether annotation or its Popup is open
Annot.line_ends	start / end appearance of line-type annotations
Annot.next	link to the next annotation
Annot.opacity	the annot's transparency
Annot.parent	page object of the annotation
Annot.popup_rect	rectangle of the annotation's Popup
Annot.popup_xref	the PDF *xref* number of the annotation's Popup
Annot.rect	rectangle containing the annotation
Annot.type	type of the annotation
Annot.vertices	point coordinates of Polygons, PolyLines, etc.
Annot.xref	the PDF *xref* number

Class API

class Annot

get_pixmap(*matrix=pymupdf.Identity*, *dpi=None*, *colorspace=pymupdf.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 `pymupdf.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 *Document.reload_page()* the page first via `page = doc.reload_page(page)`.
- The pixmap will have *"premultiplied"* pixels if `alpha=True`. To learn about some background, e.g. look for "Premultiplied alpha" in this online glossary.

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*.

get_textpage(*clip=None*, *flags=3*)

Create a *TextPage* for the annotation.

Parameters

- **flags** (*int*) – indicator bits controlling the content available for subsequent text extractions and searches – see the parameter of *Annot.get_text()*.
- **clip** (*rect-like*) – restrict extracted text to this area.

Returns

TextPage

- v1.25.5: fixed `clip` arg.

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

- Some symbols have an interior area (diamonds, circles, squares, etc.). These areas are filled with the fill color or the stroke color, depending on the annotation type.

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.

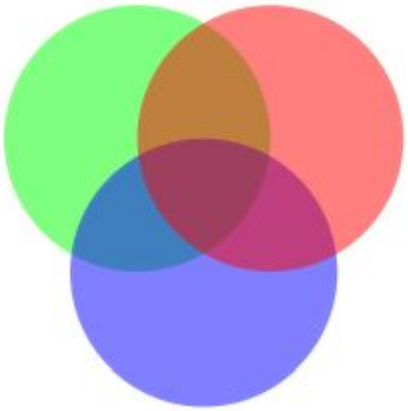
Parameters
rect (*rect_like*) – the desired rectangle.

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`.

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.

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 points. So please make sure that the following inequality is true: `pymupdf.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. This method only affects certain annotation types[2] and will lead to a message on Python's `sys.stderr` in other cases. No exception will be raised, but `False` will be returned.

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.

[2] Only the following annotation types support method *Annot.set_rect()*: Text, FreeText, Square, Circle, Redact, Stamp, Caret, FileAttachment, Sound, and Movie.

> **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, 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=None*, *style=None*, *dashes=None*, *clouds=None*)

- Changed in version 1.16.9: Allow specification without using a dictionary. The direct parameters are used if *border* is not a dictionary.
- Changed in version 1.22.5: Support of the "cloudy" border effect.

PDF only: Change border width, dashing, style and cloud effect. See the *Annot.border* attribute for more details.

Parameters

- **border** (*dict*) – a dictionary as returned by the *border* property, with keys *"width"* (*float*), *"style"* (*str*), *"dashes"* (*sequence*) and *clouds* (*int*). Omitted keys will leave the resp. property unchanged. Set the border argument to `None` (the default) to use the other arguments.
- **width** (*float*) – A non-negative value will change the border line width.
- **style** (*str*) – A value other than `None` will change this border property.
- **dashes** (*sequence*) – All items of the sequence must be integers, otherwise the parameter is ignored. To remove dashing use: `dashes=[]`. If dashes is a non-empty sequence, "style" will automatically be set to "D" (dashed).
- **clouds** (*int*) – A value `>= 0` will change this property. Use `clouds=0` to remove the cloudy appearance completely. Only annotation types 'Square', 'Circle', and 'Polygon' are supported with this property.

set_flags(*flags*)

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

Parameters

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

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

Changes the "stroke" and "fill" colors for supported annotation types – not all annotation types accept both. **Do not use this method at all for FreeText annotations** because it has its special conventions to deal with up to three colors (border, fill, text).

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.

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 <= 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 *fontsize* of the text. 'FreeText' annotations only.
- **text_color** (*sequence,float*) – change the text color. 'FreeText' annotations only. This has the same effect as `border_color`. Note that the text color of rich-text annotations cannot be changed at all because it is set by HTML / CSS syntax and part of the text itself.
- **border_color** (*sequence,float*) – change the border color. 'FreeText' annotations only. This has the same effect as `text_color`.

- **fill_color** (*sequence*, *float*) – the fill color.
- **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

This method is the only way to change the colors of a FreeText annotation. You cannot use *Annot.set_colors()* for this purpose. But be aware that for rich-text annotations, the text color is never changed. The text color is set by the `text_color` entry of the `info` dictionary. This is a limitation of and not a bug.

Caution

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.

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.

[1] Rotating an annotation also changes its rectangle. Depending on how the annotation was defined, the original rectangle is **cannot be reconstructed** by setting the rotation value to zero again and will be lost.

`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 value 0 <= 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. For information items 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 <= float <= 1* specifying the "stroke" and the interior ("fill") colors. The stroke color is used for borders and everything that is actively painted or written ("stroked"). The fill color is used for the interior of objects like line ends, circles and squares. The lengths of these lists implicitly determine the colorspaces used: 1 = GRAY, 3 = RGB, 4 = CMYK. So "[1.0, 0.0, 0.0]" stands for RGB color red. Both lists can be empty if no color is specified. Be aware about some potentially unexpected cases:

- The color of Highlight annotations is a **stroke** color, contrary to intuition.
- The color if FreeText annotations is a **stroke** color, but appears as the color that fills the rectangle and any applicable line end symbols. Text color and border color cannot be accessed at all.

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

rect_delta

A tuple of four floats representing the `/RD` entry of the annotation. The four numbers describe the numerical differences (left, top, -right, -bottom) between two rectangles: the *rect* of the annotation and a rectangle contained within that rectangle. If the entry is missing, this property is `(0, 0, 0, 0)`. If the annotation border is a normal, straight line, these numbers are typically border width divided by 2. If the annotation has a "cloudy" border, you will see the breadth of the cloud semi-circles here. In general, the numbers need not be identical. To compute the inner rectangle do `a.rect + a.rect_delta`.

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 dashing 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 line surrounding the annotation, **"D"** (Dashed) = dashed line 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.
- *clouds* – an integer indicating a "cloudy" border, where `n` is an integer `-1 <= n <= 2`. A value `n = 0` indicates a straight line (no clouds), 1 means small and 2 means large semi-circles, mimicking the cloudy appearance. If -1, then no specification is present.

Return type
dict

27.1.1 Annotation Icons in MuPDF

This is a list of icons referenceable 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'.

27.1.2 Example

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

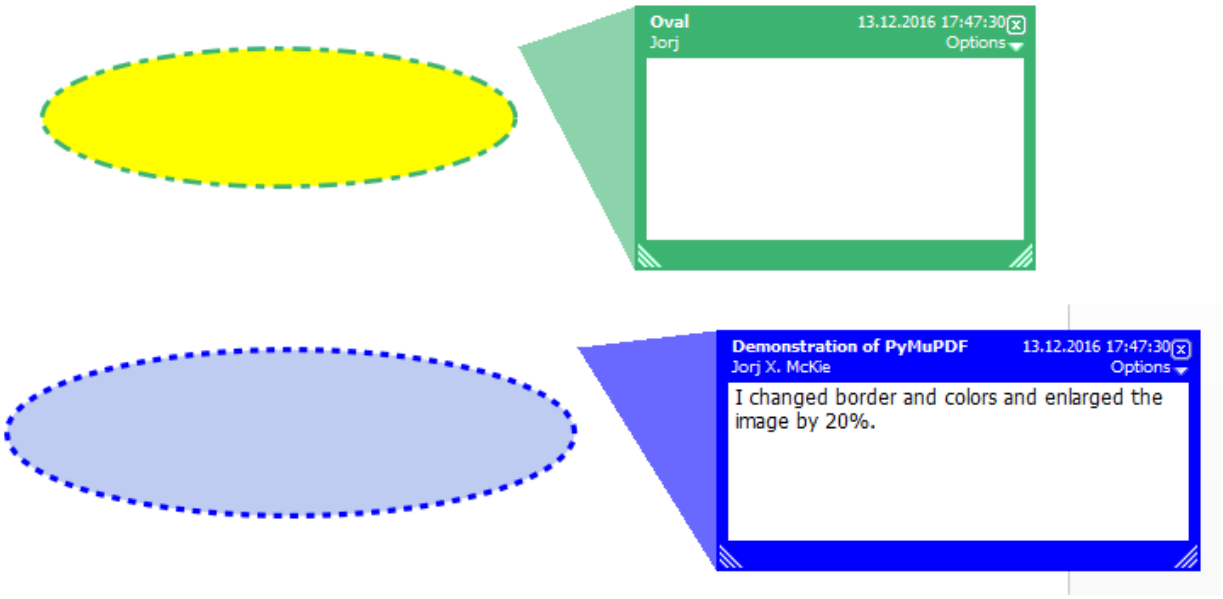
```
doc = pymupdf.open("circle-in.pdf")
page = doc[0]                           # page 0
annot = page.first_annot                 # get the annotation
annot.set_border(dashes=[3])            # set dashes to "3 on, 3 off ..."

# 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
annot.update()                          # update the annot's appearance
```

(continues on next page)

(continued from previous page)

```
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):



This documentation covers all versions up to 1.28.2.

27.2 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 and as an *option when opening files* to specify where to look for fonts, images and other resources.

Method / Attribute	Short Description
Archive.add()	add new data to the archive
Archive.has_entry()	check if given name is a member
Archive.read_entry()	read the data given by the name
Archive.entry_list	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 pymupdf
>>> 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 = pymupdf.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'}]
>>>
```

This documentation covers all versions up to 1.28.2.

27.3 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: *pymupdf.csCMYK.name = 'DeviceCMYK'*.

Type
str

n

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

type
int

Predefined Colorspaces

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

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

This documentation covers all versions up to 1.28.2.

27.4 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
run()	Run a display list through a device.
get_pixmap()	generate a pixmap
get_textpage()	generate a text page
rect	mediabox of the display list

Class API

class DisplayList

__init__(*self*, *mediabox*)

Create a new display list.

Parameters
mediabox (*Rect*) – The page's rectangle.

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=pymupdf.Identity*, *colorspace=pymupdf.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 *Font Properties*.

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

This documentation covers all versions up to 1.28.2.

27.5 Document

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

There is an alias *open()* for this class, i.e. `pymupdf.Document(...)` and `pymupdf.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 laid out 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
Document.add_layer()	PDF only: make new optional content configuration
Document.add_ocg()	PDF only: add new optional content group
Document.apply_css()	Markdown only: apply CSS stylesheet to Markdown content
Document.authenticate()	gain access to an encrypted document
Document.bake()	PDF only: make annotations / fields permanent content
Document.can_save_incrementally()	check if incremental save is possible
Document.chapter_page_count()	number of pages in chapter
Document.close()	close the document
Document.convert_to_pdf()	write a PDF version to memory
Document.copy_page()	PDF only: copy a page reference
Document.del_toc_item()	PDF only: remove a single TOC item
Document.delete_page()	PDF only: delete a page
Document.delete_pages()	PDF only: delete multiple pages
Document.embfile_add()	PDF only: add a new embedded file from buffer
Document.embfile_count()	PDF only: number of embedded files
Document.embfile_del()	PDF only: delete an embedded file entry
Document.embfile_get()	PDF only: extract an embedded file buffer
Document.embfile_info()	PDF only: metadata of an embedded file
Document.embfile_names()	PDF only: list of embedded files
Document.embfile_upd()	PDF only: change an embedded file
Document.extract_font()	PDF only: extract a font by *xref*
Document.extract_image()	PDF only: extract an embedded image by *xref*
Document.ez_save()	PDF only: *Document.save()* with different defaults
Document.find_bookmark()	retrieve page location after laid out document

continues on next page

Table 2 – continued from previous page

Method / Attribute	Short Description
Document.fullcopy_page()	PDF only: duplicate a page
Document.get_layer()	PDF only: lists of OCGs in ON, OFF, RBGroups
Document.get_layers()	PDF only: list of optional content configurations
Document.get_oc()	PDF only: get OCG /OCMD xref of image / form xobject
Document.get_ocgs()	PDF only: info on all optional content groups
Document.get_ocmd()	PDF only: retrieve definition of an *OCMD*
Document.get_page_fonts()	PDF only: list of fonts referenced by a page
Document.get_page_images()	PDF only: list of images referenced by a page
Document.get_page_labels()	PDF only: list of page label definitions
Document.get_page_numbers()	PDF only: get page numbers having a given label
Document.get_page_pixmap()	create a pixmap of a page by page number
Document.get_page_text()	extract the text of a page by page number
Document.get_page_xobjects()	PDF only: list of XObjects referenced by a page
Document.get_sigflags()	PDF only: determine signature state
Document.get_toc()	extract the table of contents
Document.get_xml_metadata()	PDF only: read the XML metadata
Document.has_annots()	PDF only: check if PDF contains any annots
Document.has_links()	PDF only: check if PDF contains any links
Document.insert_page()	PDF only: insert a new page
Document.insert_pdf()	PDF only: insert pages from another PDF
Document.insert_file()	PDF only: insert pages from arbitrary document
Document.journal_can_do()	PDF only: which journal actions are possible
Document.journal_enable()	PDF only: enables journalling for the document
Document.journal_load()	PDF only: load journal from a file
Document.journal_op_name()	PDF only: return name of a journalling step
Document.journal_position()	PDF only: return journalling status
Document.journal_redo()	PDF only: redo current operation
Document.journal_save()	PDF only: save journal to a file
Document.journal_start_op()	PDF only: start an “operation” giving it a name
Document.journal_stop_op()	PDF only: end current operation
Document.journal_undo()	PDF only: undo current operation
Document.layer_ui_configs()	PDF only: list of optional content intents
Document.layout()	re-paginate the document (if supported)
Document.load_page()	read a page
Document.make_bookmark()	create a page pointer in reflowable documents
Document.move_page()	PDF only: move a page to different location in doc
Document.need_appearances()	PDF only: get/set `/NeedAppearances` property
Document.new_page()	PDF only: insert a new empty page
Document.next_location()	return (chapter, pno) of following page
Document.outline_xref()	PDF only: *xref* a TOC item
Document.page_cropbox()	PDF only: the unrotated page rectangle
Document.page_xref()	PDF only: *xref* of a page number
Document.pages()	iterator over a page range
Document.pdf_catalog()	PDF only: *xref* of catalog (root)
Document.pdf_trailer()	PDF only: trailer source
Document.prev_location()	return (chapter, pno) of preceding page
Document.rewrite_images()	PDF only: rewrite / extra compression for images
Document.recolor()	PDF only: execute *Page.recolor()* for all pages
Document.reload_page()	PDF only: provide a new copy of a page
Document.resolve_names()	PDF only: Convert destination names into a Python dict

continues on next page

Table 2 – continued from previous page

Method / Attribute	Short Description
Document.save()	PDF only: save the document
Document.saveIncr()	PDF only: save the document incrementally
Document.scrub()	PDF only: remove sensitive data
Document.search_page_for()	search for a string on a page
Document.select()	PDF only: select a subset of pages
Document.set_layer_ui_config()	PDF only: set OCG visibility temporarily
Document.set_layer()	PDF only: mass changing OCG states
Document.set_markinfo()	PDF only: set the MarkInfo values
Document.set_metadata()	PDF only: set the metadata
Document.set_oc()	PDF only: attach OCG/OCMD to image / form xobject
Document.set_ocmd()	PDF only: create or update an *OCMD*
Document.set_page_labels()	PDF only: add/update page label definitions
Document.set_pagemode()	PDF only: set the PageMode
Document.set_pagelayout()	PDF only: set the PageLayout
Document.set_toc_item()	PDF only: change a single TOC item
Document.set_toc()	PDF only: set the table of contents (TOC)
Document.set_xml_metadata()	PDF only: create or update document XML metadata
Document.subset_fonts()	PDF only: create font subsets
Document.switch_layer()	PDF only: activate OC configuration
Document.tobytes()	PDF only: writes document to memory
Document.xref_copy()	PDF only: copy a PDF dictionary to another *xref*
Document.xref_get_key()	PDF only: get the value of a dictionary key
Document.xref_get_keys()	PDF only: list the keys of object at *xref*
Document.xref_object()	PDF only: get the definition source of *xref*
Document.xref_set_key()	PDF only: set the value of a dictionary key
Document.xref_stream_raw()	PDF only: raw stream source at *xref*
Document.xref_xml_metadata()	PDF only: *xref* of XML metadata
Document.chapter_count	number of chapters
Document.FormFonts	PDF only: list of global widget fonts
Document.is_closed	has document been closed?
Document.is_dirty	PDF only: has document been changed yet?
Document.is_encrypted	document (still) encrypted?
Document.is_fast_webaccess	is PDF linearized?
Document.is_form_pdf	is this a Form PDF?
Document.is_pdf	is this a PDF?
Document.is_reflowable	is this a reflowable document?
Document.is_repaired	PDF only: has this PDF been repaired during open?
Document.last_location	(chapter, pno) of last page
Document.metadata	metadata
Document.markinfo	PDF MarkInfo value
Document.name	filename of document
Document.needs_pass	require password to access data?
Document.outline	first *Outline* item
Document.page_count	number of pages
Document.permissions	permissions to access the document
Document.pagemode	PDF PageMode value
Document.pagelayout	PDF PageLayout value
Document.version_count	PDF count of versions

Class API

class Document

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

Create 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.
- If `stream` is `None`, then a document is created from the file given by `filename`.

Parameters

- **filename** (*str,pathlib*) – A UTF-8 string or `pathlib.Path` object containing a file path. The document type is always determined from the file content. The `filetype` parameter is ignored, except when content inspection was unsuccessful. This is regularly the case for plain text types like "txt", "html", "xml" etc. with a wrong or missing file extension.
- **stream** (*bytes,bytearray,BytesIO*) – A memory area containing file data. The document type is always detected from the data content. The `filetype` parameter is ignored, except when content inspection was unsuccessful. This is regularly the case for plain text types like "txt", "html", "xml" etc.
- **filetype** (*str*) – A string specifying the type of document. This is only ever needed when file content inspection fails. Text types like "txt", "html", "xml" etc. cannot be disambiguated by their content. When such files are provided in memory or being provided with the wrong file extension, this parameter **must** be used.
- **archive** (`Archive`) – An optional *Archive* object to use as a source for resources like fonts and images. *(new in v1.28.0)*
- **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, MD or HTML, and ignored otherwise. If specified, it must be a non-empty, finite rectangle with top-left coordinates (0, 0). Together with parameter *fontsize*, each page will be accordingly laid out and hence also determine the number of pages.
- **width** (*float*) – May be used together with `height` as an alternative to `rect` to specify layout information.
- **height** (*float*) – May be used together with `width` as an alternative to `rect` to specify layout information.
- **fontsize** (*float*) – the default *fontsize* for reflowable document types. This parameter is ignored if none of the parameters `rect` or `width` and `height` are specified. Will be used to calculate the page layout.

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(pymupdf.TOOLS.mupdf_warnings())` (which will be emptied by this call, but you can also prevent this – consult *Tools.mupdf_warnings()*).

Overview of possible forms, note: *open()* is a synonym of *Document()*:

```
# from a file
doc = pymupdf.open("some.xps")
# handle wrong extension
doc = pymupdf.open("some.file", filetype="xps")  # assert expected type
doc = pymupdf.open("some.file", filetype="txt")  # treat as plain text

# from memory
doc = pymupdf.open(stream=mem_area)  # works for any supported type
doc = pymupdf.open(stream=unknown_type, filetype="txt")  # treat as plain text

# new empty PDF
doc = pymupdf.open()
doc = pymupdf.open(None)
doc = pymupdf.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.

See *supported file types* for more information.

The Document class can be also be used as a **context manager**. Exiting the content manager will close the document automatically.

```
>>> import pymupdf
>>> with pymupdf.open(...) as doc:
        for page in doc: print(f"page {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 xobject[5]. 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

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.

[5] Examples for "Form XObjects" are created by *Page.show_pdf_page()*.

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*, *locked=None*)

- New in v1.18.3
- Changed in v1.22.5: Support list of *locked* OCGs.

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*) – state of OCGs that are not mentioned in *on* or *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.
- **locked** (*list*) – a list of OCG xref number that cannot be changed by the user interface.

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,
      'intent': ['View', 'Design'],
      'name': 'Circle',
      'usage': 'Artwork'},
14: {'on': True,
      'intent': ['View', 'Design'],
      'name': 'Square',
      'usage': 'Artwork'},
15: {'on': False, 'intent': ['View'], 'name': 'Square', 'usage': 'Artwork'}}
>>>
```

layer_ui_configs()

- New in v1.18.3

Show the visibility status of optional content that is modifiable by the user interface of supporting PDF viewers.

- 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:* true if cannot be changed via user interfaces
  - *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.

Please note that visibility is **not** a property stored with the OCG. It is not even 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** (*int*,*str*) – either the sequence number of the item in list `Document.layer_configs()` or the "text" of one of these items.
- **action** (*int*) – `PDF_OC_ON` = set on (default), `PDF_OC_TOGGLE` = toggle on/off, `PDF_OC_OFF` = set off.

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 with 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 with chapter support (EPUB currently).

prev_location(*page_id*)

- New in v.1.17.0

Return the locator of the preceding page.

Parameters
page_id (*tuple*) – the current page id. This must be a tuple *(chapter, pno)* identifying an existing page.

Returns
The tuple of the preceding page, i.e. either *(chapter, pno - 1)* or the last page of the preceding chapter, **or** the empty tuple *()* if the argument was the first page. Relevant only for document types with 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 $-\infty$ < `page_id` < `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 with 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 this[3] footnote for comments on performance improvements.

`rewrite_images`(*dpi_threshold=None*, *dpi_target=0*, *quality=0*, *lossy=True*, *lossless=True*, *bitonal=True*, *color=True*, *gray=True*, *set_to_gray=False*, *options=None*)

PDF only: Walk through all images and rewrite them according to the specified parameters. This is useful for reducing file size, changing image formats, or converting color spaces.

The typical usage is extra compression of images for significantly reducing the file size of the PDF. When setting quality and the dpi parameters to positive values and accepting defaults for the rest, the following will happen:

[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.

- Lossy and lossless images will be rewritten as JPEG images (FZ_RECOMPRESS_JPEG) as far as technically possible.
- Bitonal (monochrome) images will be rewritten in FAX format (FZ_RECOMPRESS_FAX).
- Subsampling method is **FZ_SUBSAMPLE_AVERAGE** (see below).

Parameters

- **dpi_target** (*int*) – target DPI value for the resampled images. Ignored if `dpi_threshold` is `None`, otherwise must be less than `dpi_threshold` and positive.
- **dpi_threshold** (*int*) – If None (the default) no resampling takes place. Otherwise images with a DPI value larger than this will be resampled to `dpi_target` (which must be less than `dpi_threshold`).
- **quality** (*int*) – desired target JPEG quality, a value between 0 and 100. 0 means no quality change, 100 means best quality.
- **lossy** (*bool*) – include lossy image types (e.g. JPEG).
- **lossless** (*bool*) – include lossless image types (e.g. PNG).
- **bitonal** (*bool*) – include black-and-white images (e.g. FAX).
- **color** (*bool*) – include colored images.
- **gray** (*bool*) – include grayscale images.
- **set_to_gray** (*bool*) – if True, the PDF will be converted to grayscale by executing *Document.recolor()* before all image processing. Please note that this will also change text and vector graphics to grayscale – not just the images.
- **options** (*dict*) – This parameter is intended for expert users. Except `set_to_gray`, all other parameters are ignored. It must be an object prepared in the following way: `options = pymupdf.mupdf.PdfImageRewriterOptions()`. Then attributes of this object can be set to achieve fine-grained control. Following are the adjustable attributes of the `options` object and their default (do nothing) values.

```
options.bitonal_image_recompress_method = FZ_RECOMPRESS_NEVER
options.bitonal_image_recompress_quality = None
options.bitonal_image_subsample_method = FZ_SUBSAMPLE_AVERAGE
options.bitonal_image_subsample_threshold = 0
options.bitonal_image_subsample_to = 0
options.color_lossless_image_recompress_method = FZ_RECOMPRESS_NEVER
options.color_lossless_image_recompress_quality = None
options.color_lossless_image_subsample_method = FZ_SUBSAMPLE_AVERAGE
options.color_lossless_image_subsample_threshold = 0
options.color_lossless_image_subsample_to = 0
options.color_lossy_image_recompress_method = FZ_RECOMPRESS_NEVER
options.color_lossy_image_recompress_quality = None
options.color_lossy_image_subsample_method = FZ_SUBSAMPLE_AVERAGE
options.color_lossy_image_subsample_threshold = 0
options.color_lossy_image_subsample_to = 0
options.gray_lossless_image_recompress_method = FZ_RECOMPRESS_NEVER
options.gray_lossless_image_recompress_quality = None
options.gray_lossless_image_subsample_method = FZ_SUBSAMPLE_AVERAGE
options.gray_lossless_image_subsample_threshold = 0
options.gray_lossless_image_subsample_to = 0
```

(continues on next page)

(continued from previous page)

```
options.gray_lossy_image_recompress_method = FZ_RECOMPRESS_NEVER
options.gray_lossy_image_recompress_quality = None
options.gray_lossy_image_subsample_method = FZ_SUBSAMPLE_AVERAGE
options.gray_lossy_image_subsample_threshold = 0
options.gray_lossy_image_subsample_to = 0
```

The `*_recompress_method` attributes may be one of the values **FZ_RECOMPRESS_NEVER (0), FZ_RECOMPRESS_SAME (1), FZ_RECOMPRESS_LOSSLESS (2), FZ_RECOMPRESS_JPEG (3), FZ_RECOMPRESS_J2K (4), FZ_RECOMPRESS_FAX (5)**. Value FZ_RECOMPRESS_NEVER will skip this image type altogether and FZ_RECOMPRESS_SAME will not change the type. The other values will execute type conversions (as far as technically possible).

The `*_quality` values are strings of integers from "0" to "100" or `None`.

The `*_subsample_method` attributes are either **FZ_SUBSAMPLE_AVERAGE (0)** or **FZ_SUBSAMPLE_BICUBIC (1)** and refer to how a pixel value is derived from its neighboring pixels during subsampling. For some background see this Wikipedia article about bicubic interpolation.

Attributes `*_subsample_threshold` excludes images from subsampling which have a lower DPI. Participating images will be subsampled to the DPI values given by the `*_subsample_to` values. Values of 0 mean that no subsampling will take place.

The `*_subsample_threshold` values should be chosen notably larger than the `*_subsample_to` values to ensure that there are enough size savings. After all, every subsampling inevitably incurs quality losses.

An example for a good choice is `threshold=100` and `to=72`.

recolor(*components=1*)

PDF only: Change the color component counts for all object types text, images and vector graphics for all pages.

Parameters

components (*int*) – desired color space indicated by the number of color components: 1 = DeviceGRAY, 3 = DeviceRGB, 4 = DeviceCMYK.

The typical use case is 1 (DeviceGRAY) which converts the PDF to grayscale.

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.

resolve_names()

PDF only: Convert destination names into a Python dict.

Returns

A dictionary with the following layout:

- *key*: (str) the name.
- ***value*: (dict) with the following layout:**
  - ”page”: target page number (0-based). If no page number found -1.
  - ”to”: (x, y) target point on page. Currently in PDF coordinates, i.e. point (0,0) is the bottom-left of the page.
  - ”zoom”: (float) the zoom factor.
  - ”dest”: (str) only present if the target location on the page has not been provided as “/XYZ” or if no page number was found.

Examples:

```
{
    '__bookmark_1': {'page': 0, 'to': (0.0, 541.0), 'zoom': 0.0},
    '__bookmark_2': {'page': 0, 'to': (0.0, 481.45), 'zoom': 0.0},
}
```

or:

```
{
    '21154a7c20684ceb91f9c9adc3b677c40': {'page': -1, 'dest': '/XYZ␣
↪15.75 1486 0'},
    ...
}
```

All names found in the catalog under keys “/Dests” and “/Names/Dests” are included.

- New in v1.23.6

page_cropbox(*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

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 `-∞ < start < 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 `-∞ < stop <= 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 start < stop and -1 if start > 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 = pymupdf.open("some.xps")
>>> pdfbytes = xps.convert_to_pdf()
>>>
>>> # either do this -->
>>> pdf = pymupdf.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 = pymupdf.open()                       # new PDF
>>> imglist = [ ... image file names ...] # e.g. a directory␣
↪listing
>>> for img in imglist:
        imgdoc=pymupdf.open(img)           # open image as a␣
↪document
        pdfbytes=imgdoc.convert_to_pdf()  # make a 1-page PDF of it
        imgpdf=pymupdf.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. 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 recovered[2].
- 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 `False`, each item of the list also contains a dictionary with *linkDest* details for each outline entry.

[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 convert.py.

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 source page number (*int*). `-1` if 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.
  - nameddest: target name if kind=4. PDF only. (New in 1.23.7.)

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 pymupdf
>>> doc=pymupdf.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')
>>> pprint(doc.xref_get_keys(-1))  # primary level keys of the␣
↪trailer
('Type', 'Index', 'Size', 'W', 'Root', 'Info', 'ID', 'Length',
```

(continues on next page)

(continued from previous page)

```
↪'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')
>>> #
>>> # 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')
```

(continues on next page)

(continued from previous page)

```
Index = ('array', '[0 8802]')
Size = ('int', '8802')
W = ('array', '[1 3 1]')
Root = ('xref', '8799 0 R')
Info = ('xref', '8800 0 R')
ID = ('array', '[<DC9D56A6277EFFD82084E64F9441E18C>
↪<DC9D56A6277EFFD82084E64F9441E18C>]')
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!

- *:data:`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 like this: `"/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
>>> pymupdf.get_pdf_str("Pay in $ or €.")
'<feff00500061007900200069006e002000240020006f0072002020ac002e>'
>>> # escapes for brackets and non-ASCII
>>> pymupdf.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 `-∞ < pno < 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, `-∞ < pno < 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

- *:data:`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, $-\infty$ < pno < 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)`

- `xref` (*int*) is the image object number
- `smask` (*int*) is the object number of its soft-mask image
- `width` (*int*) is the image width
- `height` (*int*) is the image height
- `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 object definition.

Parameters

- **pno** (*int*) – page number, 0-based, `-∞ < pno < 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 the object definition of the page. Each entry looks like:

`(xref, ext, type, basefont, name, encoding, referencer)`

- `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 true 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 make only partial use.
- Be aware that font names returned by some variants of *Page.get_text()* (respectively *TextPage* methods) need not (exactly) equal the base font name shown here. Reasons for any differences include:
  - This method always shows any subset prefixes (the pattern `ABCDEF+`), whereas text extractions do not do this by default.

- Text extractions use the base library to access the font name, which has a length cap of 31 bytes and generally interrogates the font file binary to access the name. Method `get_page_fonts()` however looks at the PDF definition source.
- Text extractions work for all supported document types in exactly the same way – not just for PDFs. Consequently they do not contain PDF-specifics.

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 `-∞ < pno < 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 or this chapter.

Parameters

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

Note

- Page numbers in the sequence need not be unique nor be in any particular order. This makes the method a versatile utility to e.g. select only the even or the odd pages or meeting some other criteria and so forth.
- On a technical level, the method will always create a new *pagetree*.
- When dealing with only a few pages, methods *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_pagelayout`(*value*)

- New in v1.22.2

PDF only: Set the `/PageLayout`.

Parameters
value (*str*) – one of the strings "SinglePage", "OneColumn", "TwoColumnLeft", "TwoColumnRight", "TwoPageLeft", "TwoPageRight". Lower case is supported.

`set_pagemode`(*value*)

- New in v1.22.2

PDF only: Set the `/PageMode`.

Parameters
value (*str*) – one of the strings "UseNone", "UseOutlines", "UseThumbs", "FullScreen", "UseOC", "UseAttachments". Lower case is supported.

`set_markinfo`(*value*)

- New in v1.22.2

PDF only: Set the `/MarkInfo` values.

Parameters
value (*dict*) – a dictionary like this one: `{"Marked": False, "UserProperties": False, "Suspects": False}`. This dictionary contains information about the usage of Tagged PDF conventions. For details please see the PDF specifications.

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.

Changed in v1.23.8: Destination 'to' coordinates should now be in the same coordinate system as those returned by *get_toc()* (internally they are now transformed with `page.cropbox` and `page.rotation_matrix`). So for example `set_toc(get_toc())` now gives unchanged destination 'to' coordinates.

outline_xref(*idx*)

- New in v1.17.7

PDF only: Return the *xref* of the outline item. This is mainly used for internal purposes.

Parameters
idx (*int*) – 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 structure 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 <= pno <= 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 pymupdf
>>> doc=pymupdf.open("SWIGDocumentation.pdf")
>>> toc = doc.get_toc(False)  # we need the detailed TOC
```

(continues on next page)

(continued from previous page)

```
>>> # 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.

bake(*, *annots=True*, *widgets=True*)

PDF only: Convert annotations and / or widgets to become permanent parts of the pages. The PDF **will be changed** by this method. If *widgets* is `True`, the document will also no longer be a "Form PDF".

All pages will look the same, but will no longer have annotations, respectively fields. The visible parts will be converted to standard text, vector graphics or images as required.

The method may thus be a viable **alternative for PDF-to-PDF conversions** using *Document.convert_to_pdf()*.

Please consider that annotations are complex objects and may consist of more data "underneath" their visual appearance. Examples are "Text" and "FileAttachment" annotations. When "baking in" annotations / widgets with this method, all this underlying information (attached files, comments, associated PopUp annotations, etc.) will be lost and be removed on next garbage collection.

Use this feature for instance for *Page.show_pdf_page()* (which supports neither annotations nor widgets) when the source pages should look exactly the same in the target.

Parameters

- **annots** (*bool*) – convert annotations.
- **widgets** (*bool*) – convert fields / widgets. After execution, the document will no longer be a "Form PDF".

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.

repair()

Repair document.

- Slow for large documents.
- Does nothing on non-PDF documents.
- New in v1.27.0

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.

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 OCRed text and invisible text[7].
- **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*, *use_objstms=0*, *compression_effort=0*, *raise_on_repair=False*)

PDF documents are saved in their **current state**.

Non-PDF documents are saved in PDF format. *(new in v1.28.0)*

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.

[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.

- **clean** (*bool*) – Clean and sanitize content streams[1]. Corresponds to "mutool clean -sc".
- **deflate** (*int*) – Deflate (compress) uncompressed streams. See: *Compressing Files* for full details on this parameter.
- **deflate_images** (*bool*) – *(new in v1.18.3)* Deflate (compress) uncompressed image streams[4].
- **deflate_fonts** (*bool*) – *(new in v1.18.3)* Deflate (compress) uncompressed fontfile streams[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" and "use_objstms".
- **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.
- **use_objstms** (*int*) – *(new in v1.24.0)* compression option that converts eligible PDF object definitions to information that is stored in some other object's `stream` data. Depending on the `deflate` parameter value, the converted object definitions will be compressed – which can lead to very significant file size reductions.

[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.

[4] These parameters cause separate handling of stream categories: use it together with `expand` to restrict decompression to streams other than images / fontfiles.

See: *Compressing Files* for full details on this parameter.

- **compression_effort** (*int*) –
  - 0 for default
  - 1 for minimum effort.
  - 100 for maximum effort.

  See: *Compressing Files* for full details on this parameter.
- **raise_on_repair** (*bool*) – *(new in v1.27.0)* If true we raise an exception if the save caused a repair. This is useful because repairs can cause changes to be lost.

  Also see `Document.repair()`.

Warning

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.

Note

File size reduction

1. Use the save options like `garbage=3|4, deflate=True, use_objstms=True|1`. Do not touch the default values `expand=False|0, clean=False|0, incremental=False|0, linear=False|0`. This is a "lossless" file size reduction. There is a convenience version of this method with these values set by default, `Document.ez_save()` – please see below.
2. "Lossy" file size reduction in essence must give up something with respect to images, like (a) remove all images (b) replace images by their grayscale versions (c) reduce image resolutions. Find examples in the PyMuPDF Utilities "replace-image" folder.

See: *Compressing Files* for more.

ez_save(**args*, ***kwargs*)

- New in v1.18.11

PDF only: The same as `Document.save()` but with changed defaults `deflate=1, garbage=3, use_objstms=True`.

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*, *use_objstms=0*)

- Changed in v1.18.7
- Changed in v1.19.0
- Changed in v1.24.1

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 pdfrw.

(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 $-\infty$ < `pno` < `page_count` is acceptable.

insert_pdf(*docsrc*, *, *from_page=-1*, *to_page=-1*, *start_at=-1*, *rotate=-1*, *links=True*, *annots=True*, *widgets=True*, *join_duplicates=False*, *show_progress=0*, *final=1*)

PDF only: Copy the page range **[from_page, to_page]** (including both) of PDF document *docsrc* into the current one. Inserts will start with page number *start_at*. Value -1 indicates default values. All pages thus copied will be rotated as specified. Links, annotations and widgets 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`. *Named* links (*LINK_NAMED*) and internal links to outside the copied page range are **always excluded**.
- **annots** (*bool*) – choose whether annotations should be included in the copy.
- **widgets** (*bool*) – choose whether annotations should be included in the copy. If `True` and at least one of the source pages contains form fields, the target PDF will be turned into a Form PDF (if not already being one).
- **join_duplicates** (*bool*) – *(New in version 1.25.5)* Choose how to handle duplicate root field names in the source pages. This parameter is ignored if `widgets=False`.

  Default is `False` which will add unifying strings to the name of those source root fields which have a duplicate in the target. For instance, if "name" already occurs in the target, the source widget's name will be changed to "name [text]" with a suitably chosen string "text".

If `True`, root fields with duplicate names in source and target will be converted to so-called "Kids" of a "Parent" object (which lists all kid widgets in a PDF array). This will effectively turn those kids into instances of the "same" widget: if e.g. one of the kids is changed, then all its instances will automatically inherit this change – no matter on which page they happen to be displayed.

- **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. This is a page-based method. Document-level information of source documents is therefore mostly ignored. Examples include Optional Content, Embedded Files, `StructureElem`, table of contents, page labels, metadata, named destinations (and other named entries) and some more.
2. If `from_page > to_page`, pages will be **copied in reverse order**. If `0 <= from_page == to_page`, then one page will be copied.
3. `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 join.py in the *examples* directory: it can join PDF documents and at the same time piece together respective parts of the tables of contents.

insert_file(*infile*, *from_page=-1*, *to_page=-1*, *start_at=-1*, *rotate=-1*, *links=True*, *annots=True*, *show_progress=0*, *final=1*)

- New in v1.22.0

PDF only: Add an arbitrary supported document to the current PDF. Opens "infile" as a document, converts it to a PDF and then invokes *Document.insert_pdf()*. Parameters are the same as for that method. Among other things, this features an easy way to append images as full pages to an output PDF.

Parameters

infile (*multiple*) – the input document to insert. May be a filename specification as is valid for creating a *Document* or a *Pixmap*.

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

PDF only: Insert an empty page.

Parameters

- **pno** (*int*) – page number index (zero-indexed) at which to insert page. 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. Be aware that the page numbers of pages after the inserted one

will have changed after method execution. For the same reason, **all existing page objects will be invalidated.** Using them will lead to exceptions.

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 index (zero-indexed) at which to insert page. 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 $-\infty$ < `pno` < `page_count`.

- 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*, ***kwds*)

- 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 then "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
- `description` – (*str*) description
- `size` – (*int*) original file size
- `length` – (*int*) compressed file length
- `creationDate` – (*str*) date-time of item creation in PDF format
- `modDate` – (*str*) date-time of last change in PDF format
- `collection` – (*int*) *xref* of the associated PDF portfolio item if any, else zero.
- `checksum` – (*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()

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 *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 v.1.16.8
- 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 this data is 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 in this PyMuPDF Utilities example.

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 = pymupdf.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 = pymupdf.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 = pymupdf.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*).

Parameters

- **xref** (*int*) – PDF object number of the font to extract.
- **info_only** (*bool*) – only return font information, not the buffer. To be used for information-only purposes, avoids allocation of large buffer areas.
- **named** (*bool*) – 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).

Return type
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 time[6] if no link, resp. no annotation is found.

subset_fonts(*verbose=False*, *fallback=False*)

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 subset of its characters.

Use this method immediately before saving the document.

Parameters

- **verbose** (*bool*) – write various progress information to sysout. This currently only has an effect if `fallback` is `True`.
- **fallback** (*bool*) – if `True` use the deprecated algorithm that makes use of package fontTools (which hence must be installed). If using the recommended value `False` (default), MuPDF's native function is used – which is **very much faster** and can subset a broader range of font types. Package fontTools is not required then.

The greatest benefit can be achieved when creating new PDFs using large fonts like is typical for Asian scripts. When using the *Story* class or method *Page.insert_htmlbox()*, multiple fonts may automatically be included – without the programmer becoming aware of it.

[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.

In all these cases, the set of actually used unicodes mostly is very small compared to the number of glyphs available in the used fonts. Using this method can easily reduce the embedded font binaries by two orders of magnitude – from several megabytes down to a low two-digit kilobyte amount.

Creating font subsets leaves behind a large number of large, now unused PDF objects ("ghosts"). Therefore, make sure to compress and garbage-collect when saving the file. We recommend to use *Document.ez_save()*.

- New in v1.18.7
- Changed in v1.18.9
- Changed in v1.24.2 use native function of MuPDF.

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.

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.

`apply_css(`*css*, *append=True*`)`

- New in v1.28.0

Apply CSS styles to the document. This is a global operation, which means that the styles will be applied to all pages and all elements of the document. The CSS syntax is the same as for HTML documents, but only a subset of CSS properties is supported.

Parameters

- **css** (*str*) – a string containing the CSS styles to be applied.
- **append** (*bool*) – whether to append the new styles to existing ones (if any) or to replace them.

> **Note**
>
> This method is primarily intended for use with *Markdown documents*.

`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, HTML or Markdown). 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

`is_fast_webaccess`

- New in v1.22.2

`True` if PDF is in linearized format. `False` for non-PDF documents.

Type
bool

`markinfo`

- New in v1.22.2

A dictionary indicating the `/MarkInfo` value. If not specified, the empty dictionary is returned. If not a PDF, `None` is returned.

Type
dict

`pagemode`

- New in v1.22.2

A string containing the `/PageMode` value. If not specified, the default "UseNone" is returned. If not a PDF, `None` is returned.

Type
str

`pagelayout`

- New in v1.22.2

A string containing the `/PageLayout` value. If not specified, the default "SinglePage" is returned. If not a PDF, `None` is returned.

Type
str

`version_count`

- New in v1.22.2

An integer counting the number of versions present in the document. Zero if not a PDF, otherwise the number of incremental saves plus one.

Type
int

`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 & pymupdf.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 *needPass=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 interval relative to GMT) containing a sign ('+' or '-'), the hour (*hh*), and the minute (*'mm'*, note the apostrophes!).
- 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*.

27.5.1 `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 pymupdf
>>> doc=pymupdf.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.del_xml_metadata()    # clear any XML metadata
>>> doc.save("anonymous.pdf", garbage = 4)       # save anonymized doc
```

27.5.2 `set_toc()` Demonstration

This shows how to modify or add a table of contents. Also have a look at import.py and export.py in the examples directory.

```
>>> import pymupdf
>>> doc = pymupdf.open("test.pdf")
```

(continues on next page)

(continued from previous page)

```
>>> 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]
```

27.5.3 `insert_pdf()` Examples

(1) Concatenate two documents including their TOCs:

```
>>> doc1 = pymupdf.open("file1.pdf")          # must be a PDF
>>> doc2 = pymupdf.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 join.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)
```

27.5.4 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
```

(continues on next page)

(continued from previous page)

```
        pix = pymupdf.Pixmap(doc, xref)   # make pixmap from image
        if pix.n - pix.alpha < 4:       # can be saved as PNG
            pix.save(f"p{i}-{xref}.png")
        else:                           # CMYK: must convert first
            pix0 = pymupdf.Pixmap(pymupdf.csRGB, pix)
            pix0.save(f"p{i}-{xref}.png")
            pix0 = None                 # free Pixmap resources
        pix = None                      # free Pixmap resources
```

Rotate all pages of a PDF:

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

This documentation covers all versions up to 1.28.2.

27.6 DocumentWriter

- New in v1.21.0

This class represents a utility which can output various *document types supported by PyMuPDF*.

In 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
DocumentWriter.begin_page()	start a new output page
DocumentWriter.end_page()	finish the current output page
DocumentWriter.close()	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.

This class can also be used as a Python context manager.

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*.

This documentation covers all versions up to 1.28.2.

27.7 Font

- New in v1.16.18

This class represents a font as defined in (`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
glyph_advance()	Width of a character
glyph_bbox()	Glyph rectangle
glyph_name_to_unicode()	Get unicode from glyph name
has_glyph()	Return glyph id of unicode
text_length()	Compute string length
char_lengths()	Tuple of char widths of a string
unicode_to_glyph_name()	Get glyph name of a unicode
valid_codepoints()	Array of supported unicodes
ascender	Font ascender
descender	Font descender
bbox	Font rectangle
buffer	Copy of the font's binary image
flags	Collection of font properties
glyph_count	Number of supported glyphs
name	Name of font
is_bold	`True` if bold
is_monospaced	`True` if mono-spaced
is_serif	`True` if serif, `False` if sans-serif
is_italic	`True` if italic

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 system[1].
- **fontbuffer** (*bytes*, *bytearray*, *io.BytesIO*) – a fontfile loaded in memory[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 serifed font.

Returns

a 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.
fontbuffer?	Create font from buffer, exception if failure.
ordering>=0	Create universal font, always succeeds.
fontname?	Create a Base-14 font, universal font, or font provided by pymupdf-fonts. See table below.

[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.

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, Hewbrew 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 pymupdf
>>> font = pymupdf.Font("math")
>>> vuc = font.valid_codepoints()
>>> for i in vuc:
>>>     print(f"{i:04X} {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)
...
```

(continues on next page)

[2] The built-in Python module `array` has been chosen for its speed and low memory requirement.

(continued from previous page)

```
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**.

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 = 0xFFFD 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 = pymupdf.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 `pymupdf.unicode_to_glyph_name()` and resp. `pymupdf.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 = pymupdf.Font("helv")
>>> text = "PyMuPDF"
>>> font.text_length(text)
50.115999937057495
>>> pymupdf.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 ascender typography 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 and lower case "a".

Return type
float

descender

- New in v1.18.0

The descender value of the font, see descender typography 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_bold

is_italic

is_monospaced

is_serif

A number of attributes with obvious meanings. Reflect some values of the *Font.flags* dictionary.

Return type
bool

This documentation covers all versions up to 1.28.2.

27.8 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 *pymupdf.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 = pymupdf.Matrix(1, 0, 0, 1, 0, 0)  # specify the values
>>> m = pymupdf.Matrix(1, 1)              # use scaling by factor 1
>>> m = pymupdf.Matrix(0)                 # use rotation by zero degrees
>>> m = pymupdf.Matrix(pymupdf.Identity)     # make a copy of Identity
```

This documentation covers all versions up to 1.28.2.

27.9 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 *bl*
IRect.bottom_right	bottom right point, synonym *br*
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 *tl*
IRect.top_right	top_right point, synonym *tr*
IRect.quad	*Quad* made from rectangle corners
IRect.width	width of the rectangle
IRect.x0	X-coordinate of the top left corner
IRect.x1	X-coordinate of the bottom right corner
IRect.y0	Y-coordinate of the top left corner
IRect.y1	Y-coordinate of the bottom right corner

Class API

class IRect

__init__(*self*)

__init__(*self*, *x0*, *y0*, *x1*, *y1*)

__init__(*self*, *irect*)

__init__(*self*, *sequence*)

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

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

If sequence is specified, it must be a Python sequence type of 4 numbers (see *Using Python Sequences as Arguments in PyMuPDF*). Non-integer numbers will be truncated, non-numeric 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*.

This documentation covers all versions up to 1.28.2.

27.10 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
Link.set_border()	modify border properties
Link.set_colors()	modify color properties
Link.set_flags()	modify link flags
Link.border	border characteristics
Link.colors	border line color
Link.dest	points to destination details
Link.is_external	checks if the link is an external destination
Link.flags	link annotation flags
Link.next	points to next link
Link.rect	clickable area in untransformed coordinates
Link.uri	link destination
Link.xref	*xref* number of the entry

Class API

class Link

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 <= float <= 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

`is_external`

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 `is_external`:

- `is_external` is true: `uri` points to some target outside the current PDF, which may be an internet resource (`uri` starts with `http://` or similar), another file (`uri` starts with "file:" or "file://") or some other service like an e-mail address (`uri` starts with `mailto:`).
- `is_external` is false: `uri` will be `None` or point 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, for example "../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

This documentation covers all versions up to 1.28.2.

27.11 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
linkDest.dest	destination
linkDest.fileSpec	file specification (path, filename)
linkDest.flags	descriptive flags
linkDest.isMap	is this a MAP?
linkDest.isUri	is this a URI?
linkDest.kind	kind of destination
linkDest.lt	top left coordinates
linkDest.named	name if named destination
linkDest.newWindow	name of new window
linkDest.page	page number
linkDest.rb	bottom right coordinates
linkDest.uri	URI

Class API

`class linkDest`

`dest`

Target destination name if *linkDest.kind* is *LINK_GOTOR* and *linkDest.page* is *-1*.

Type
str

`fileSpec`

Contains the filename and path this link points to, if *linkDest.kind* is *LINK_GOTOR* or *LINK_LAUNCH*.

Type
str

`flags`

A bitfield describing the validity and meaning of the different aspects of the destination. As far as possible, link destinations are constructed such that e.g. *linkDest.lt* and *linkDest.rb* can be treated as defining a bounding box. But the flags indicate which of the values were actually specified, see *Link Destination Flags*.

Type
int

isMap

This flag specifies whether to track the mouse position when the URI is resolved. Default value: False.

Type
bool

isUri

Specifies whether this destination is an internet resource (as opposed to e.g. a local file specification in URI format).

Type
bool

kind

Indicates the type of this destination, like a place in this document, a URI, a file launch, an action or a place in another file. Look at *Link Destination Kinds* to see the names and numerical values.

Type
int

lt

The top left *Point* of the destination.

Type
Point

named

This destination refers to some named action to perform (e.g. a javascript, see *Adobe PDF 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

This documentation covers all versions up to 1.28.2.

27.12 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
Matrix.prerotate()	perform a rotation
Matrix.prescale()	perform a scaling
Matrix.preshear()	perform a shearing (skewing)
Matrix.pretranslate()	perform a translation (shifting)
Matrix.concat()	perform a matrix multiplication
Matrix.invert()	calculate the inverted matrix
Matrix.norm()	the Euclidean norm
Matrix.a	zoom factor X direction
Matrix.b	shearing effect Y direction
Matrix.c	shearing effect X direction
Matrix.d	zoom factor Y direction
Matrix.e	horizontal shift
Matrix.f	vertical shift
Matrix.is_rectilinear	true if rect corners will remain rect corners

Class API

class Matrix

__init__(*self*)

__init__(*self*, *zoom-x*, *zoom-y*)

__init__(*self*, *shear-x*, *shear-y*, *1*)

__init__(*self*, *a*, *b*, *c*, *d*, *e*, *f*)

__init__(*self*, *matrix*)

__init__(*self*, *degree*)

__init__(*self*, *sequence*)

Overloaded constructors.

Without parameters, the zero matrix *Matrix(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)* will be created.

*zoom-** and *shear-** specify zoom or shear values (float) and create a zoom or shear matrix, respectively.

For "matrix" a **new copy** of another matrix will be made.

Float value "degree" specifies the creation of a rotation matrix which rotates anti-clockwise.

A "sequence" must be any Python sequence object with exactly 6 float entries (see *Using Python Sequences as Arguments in PyMuPDF*).

pymupdf.Matrix(1, 1) and *pymupdf.Matrix(pymupdf.Identity)* create modifiable versions of the *Identity* matrix, which looks like *[1, 0, 0, 1, 0, 0]*.

norm()

- New in version 1.16.0

Return the Euclidean norm of the matrix as a vector.

prerotate(*deg*)

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

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

Parameters

deg (*float*) – The rotation angle in degrees (use conventional notation based on Pi = 180 degrees).

prescale(*sx*, *sy*)

Modify the matrix to scale by the zoom factors sx and sy. Has effects on attributes *a* thru *d* only: *[a, b, c, d, e, f] -> [a*sx, b*sx, c*sy, d*sy, e, f]*.

Parameters

- **sx** (*float*) – Zoom factor in X direction. For the effect see description of attribute *a*.
- **sy** (*float*) – Zoom factor in Y direction. For the effect see description of attribute *d*.

preshear(*sx*, *sy*)

Modify the matrix to perform a shearing, i.e. transformation of rectangles into parallelograms (rhomboids). Has effects on attributes *a* thru *d* only: *[a, b, c, d, e, f] -> [c*sy, d*sy, a*sx, b*sx, e, f]*.

Parameters

- **sx** (*float*) – Shearing effect in X direction. See attribute *c*.
- **sy** (*float*) – Shearing effect in Y direction. See attribute *b*.

pretranslate(*tx*, *ty*)

Modify the matrix to perform a shifting / translation operation along the x and / or y axis. Has effects on attributes *e* and *f* only: *[a, b, c, d, e, f] -> [a, b, c, d, tx*a + ty*c, tx*b + ty*d]*.

Parameters

- **tx** (*float*) – Translation effect in X direction. See attribute *e*.

- **ty** (*float*) – Translation effect in Y direction. See attribute *f*.

concat(*m1*, *m2*)

Calculate the matrix product *m1 * m2* and store the result in the current matrix. Any of *m1* or *m2* may be the current matrix. Be aware that matrix multiplication is not commutative. So the sequence of *m1*, *m2* is important.

Parameters

- **m1** (*Matrix*) – First (left) matrix.
- **m2** (*Matrix*) – Second (right) matrix.

invert(*m=None*)

Calculate the matrix inverse of *m* and store the result in the current matrix. Returns *1* if *m* is not invertible ("degenerate"). In this case the current matrix **will not change**. Returns *0* if *m* is invertible, and the current matrix is replaced with the inverted *m*.

Parameters

m (*Matrix*) – Matrix to be inverted. If not provided, the current matrix will be used.

Return type

int

a

Scaling in X-direction **(width)**. For example, a value of 0.5 performs a shrink of the **width** by a factor of 2. If a < 0, a left-right flip will (additionally) occur.

Type

float

b

Causes a shearing effect: each `Point(x, y)` will become `Point(x, y - b*x)`. Therefore, horizontal lines will be "tilt".

Type

float

c

Causes a shearing effect: each `Point(x, y)` will become `Point(x - c*y, y)`. Therefore, vertical lines will be "tilt".

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

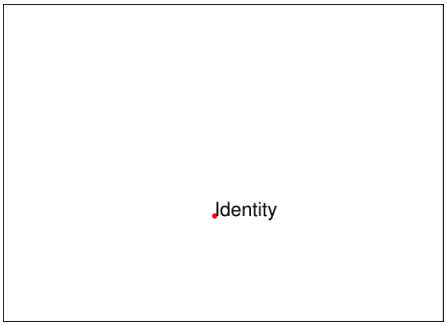
Note

- 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*.
- Matrices can be used with arithmetic operators almost like ordinary numbers: they can be added, subtracted, multiplied or divided – see chapter *Operator Algebra for Geometry Objects*.
- Matrix multiplication is **not commutative** – changing the sequence of the multiplicands will change the result in general. So it can quickly become unclear which result a transformation will yield.

27.12.1 Examples

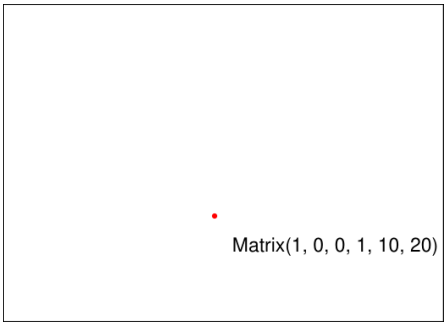
Here are examples that illustrate some of the achievable effects. All pictures show some text, inserted under control of some matrix and relative to a fixed reference point (the red dot).

1. The *Identity* matrix performs no operation.

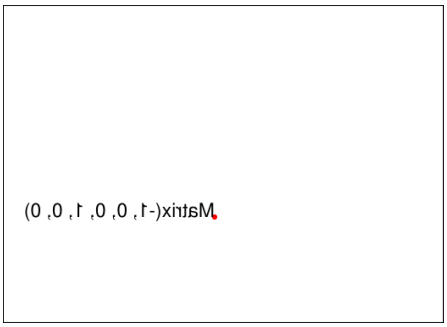


2. The scaling matrix `Matrix(2, 0.5)` stretches by a factor of 2 in horizontal, and shrinks by factor 0.5 in vertical direction.

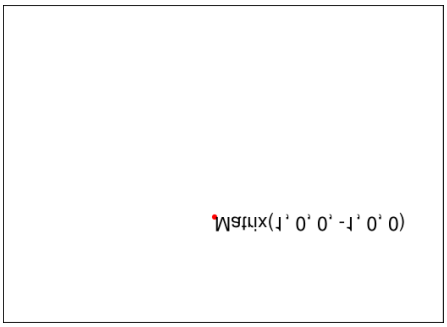
3. Attributes *Matrix.e* and *Matrix.f* shift horizontally and, respectively vertically. In the following 10 to the right and 20 down.



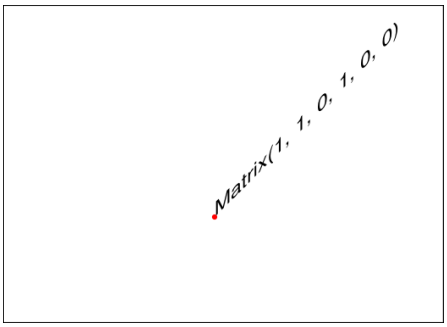
4. A negative *Matrix.a* causes a left-right flip.



5. A negative *Matrix.d* causes an up-down flip.

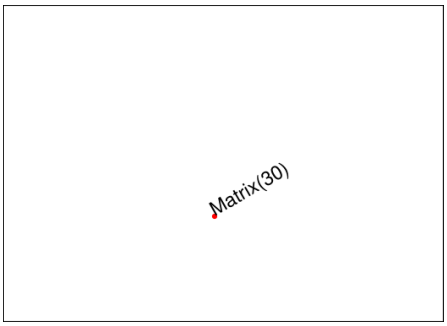


6. Attribute *Matrix.b* tilts upwards / downwards along the x-axis.



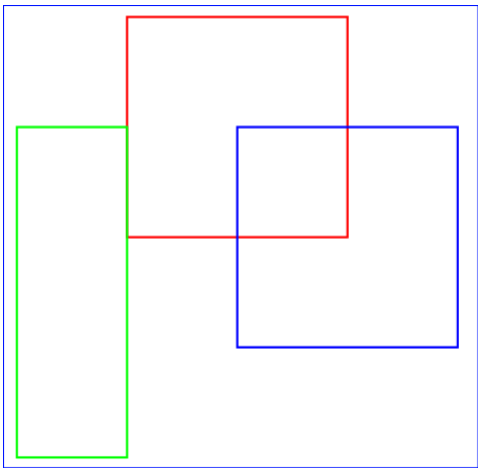
7. Attribute *Matrix.c* tilts left / right along the y-axis.

8. Matrix `Matrix(beta)` performs counterclockwise rotations for positive angles `beta`.



9. Show some effects on a rectangle:

```
import pymupdf

# just definitions and a temp PDF
RED = (1, 0, 0)
BLUE = (0, 0, 1)
GREEN = (0, 1, 0)
doc = pymupdf.open()
page = doc.new_page()

# rectangle
r1 = pymupdf.Rect(100, 100, 200, 200)

# scales down by 50% in x- and up by 50% in y-direction
mat1 = pymupdf.Matrix(0.5, 1.5)

# shifts by 50 in both directions
mat2 = pymupdf.Matrix(1, 0, 0, 1, 50, 50)

# draw corresponding rectangles
page.draw_rect(r1, color=RED)  # original
page.draw_rect(r1 * mat1, color=GREEN)  # scaled
page.draw_rect(r1 * mat2, color=BLUE)  # shifted
doc.ez_save("matrix-effects.pdf")
```



This documentation covers all versions up to 1.28.2.

27.13 Outline

The document outline (otherwise known as "bookmarks") is a property of *Document* (see *Document.outline*). If not None, it stands for the first outline item of the document. Its properties in turn define the characteristics of this item and also point to other outline items in "horizontal" or downward direction. The full tree of all outline items for e.g. a conventional table of contents (TOC) can be recovered by following these "pointers".

Method / Attribute	Short Description
Outline.down	next item downwards
Outline.next	next item same level
Outline.page	page number (0-based)
Outline.title	title
Outline.uri	string further specifying outline target
Outline.is_external	target outside document
Outline.is_open	whether sub-outlines are open or collapsed
Outline.dest	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 children.

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 property *is_external*:

- *is_external* is true: `uri` points to some target outside the current PDF, which may be an internet resource (`uri` starts with `http://` or similar), another file (`uri` starts with `file:` or `file://`) or some other service like an e-mail address (`uri` starts with `mailto:`).
- *is_external* is false: `uri` will be `None` or point 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, for example "../FixedDoc.fdoc#PG_2_LNK_1" for page number 2 (1-based) in an XPS document.

Type
str

dest

The link destination details object.

Type
linkDest

This documentation covers all versions up to 1.28.2.

27.14 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.

Note

Many times in this chapter we are using the term **coordinate**. It is of high importance to have at least a basic understanding of what that is and that you feel comfortable with the section *Coordinates*.

27.14.1 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 these coordinates **must always** be provided relative to the **unrotated** page (since v1.17.0). 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.

Releoading 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
`Page.add_caret_annot()`	PDF only: add a caret annotation
`Page.add_circle_annot()`	PDF only: add a circle annotation
`Page.add_file_annot()`	PDF only: add a file attachment annotation
`Page.add_freetext_annot()`	PDF only: add a text annotation
`Page.add_highlight_annot()`	PDF only: add a "highlight" annotation
`Page.add_ink_annot()`	PDF only: add an ink annotation
`Page.add_line_annot()`	PDF only: add a line annotation
`Page.add_polygon_annot()`	PDF only: add a polygon annotation

continues on next page

Table 3 – continued from previous page

Method / Attribute	Short Description
Page.add_polyline_annot()	PDF only: add a multi-line annotation
Page.add_rect_annot()	PDF only: add a rectangle annotation
Page.add_redact_annot()	PDF only: add a redaction annotation
Page.add_squiggly_annot()	PDF only: add a “squiggly” annotation
Page.add_stamp_annot()	PDF only: add a “rubber stamp” annotation
Page.add_strikeout_annot()	PDF only: add a “strike-out” annotation
Page.add_text_annot()	PDF only: add a comment
Page.add_underline_annot()	PDF only: add an “underline” annotation
Page.add_widget()	PDF only: add a PDF Form field
Page.annot_names()	PDF only: a list of annotation (and widget) names
Page.annot_xrefs()	PDF only: a list of annotation (and widget) xrefs
Page.annots()	return a generator over the annots on the page
Page.apply_redactions()	PDF only: process the redactions of the page
Page.clip_to_rect()	PDF only: remove page content outside a rectangle
Page.bound()	rectangle of the page
Page.cluster_drawings()	PDF only: bounding boxes of vector graphics
Page.delete_annot()	PDF only: delete an annotation
Page.delete_image()	PDF only: delete an image
Page.delete_link()	PDF only: delete a link
Page.delete_widget()	PDF only: delete a widget / field
Page.draw_bezier()	PDF only: draw a cubic Bezier curve
Page.draw_circle()	PDF only: draw a circle
Page.draw_curve()	PDF only: draw a special Bezier curve
Page.draw_line()	PDF only: draw a line
Page.draw_oval()	PDF only: draw an oval / ellipse
Page.draw_polyline()	PDF only: connect a point sequence
Page.draw_quad()	PDF only: draw a quad
Page.draw_rect()	PDF only: draw a rectangle
Page.draw_sector()	PDF only: draw a circular sector
Page.draw_squiggle()	PDF only: draw a squiggly line
Page.draw_zigzag()	PDF only: draw a zig-zagged line
Page.find_tables()	locate tables on the page
Page.get_drawings()	get vector graphics on page
Page.get_fonts()	PDF only: get list of referenced fonts
Page.get_image_bbox()	PDF only: get bbox and matrix of embedded image
Page.get_image_info()	get list of meta information for all used images
Page.get_image_rects()	PDF only: improved version of Page.get_image_bbox()
Page.get_images()	PDF only: get list of referenced images
Page.get_label()	PDF only: return the label of the page
Page.get_links()	get all links
Page.get_pixmap()	create a page image in raster format
Page.get_svg_image()	create a page image in SVG format
Page.get_text()	extract the page’s text
Page.get_textbox()	extract text contained in a rectangle
Page.get_textpage_ocr()	create a TextPage with OCR for the page
Page.get_textpage()	create a TextPage for the page
Page.get_xobjects()	PDF only: get list of referenced xobjects
Page.insert_font()	PDF only: insert a font for use by the page
Page.insert_image()	PDF only: insert an image
Page.insert_link()	PDF only: insert a link

continues on next page

Table 3 – continued from previous page

Method / Attribute	Short Description
Page.insert_text()	PDF only: insert text
Page.insert_htmlbox()	PDF only: insert html text in a rectangle
Page.insert_textbox()	PDF only: insert a text box
Page.links()	return a generator of the links on the page
Page.load_annot()	PDF only: load a specific annotation
Page.load_widget()	PDF only: load a specific field
Page.load_links()	return the first link on a page
Page.new_shape()	PDF only: create a new *Shape*
Page.recolor()	PDF only: change the colorspace of objects
Page.remove_rotation()	PDF only: set page rotation to 0
Page.replace_image()	PDF only: replace an image
Page.search_for()	search for a string
Page.set_artbox()	PDF only: modify `/ArtBox`
Page.set_bleedbox()	PDF only: modify `/BleedBox`
Page.set_cropbox()	PDF only: modify the `cropbox` (visible page)
Page.set_mediabox()	PDF only: modify `/MediaBox`
Page.set_rotation()	PDF only: set page rotation
Page.set_trimbox()	PDF only: modify `/TrimBox`
Page.show_pdf_page()	PDF only: display PDF page image
Page.update_link()	PDF only: modify a link
Page.widgets()	return a generator over the fields on the page
Page.write_text()	write one or more *TextWriter* objects
Page.cropbox_position	displacement of the `cropbox`
Page.cropbox	the page's `cropbox`
Page.artbox	the page's `/ArtBox`
Page.bleedbox	the page's `/BleedBox`
Page.trimbox	the page's `/TrimBox`
Page.derotation_matrix	PDF only: get coordinates in unrotated page space
Page.first_annot	first *Annot* on the page
Page.first_link	first *Link* on the page
Page.first_widget	first widget (form field) on the page
Page.mediabox_size	bottom-right point of `mediabox`
Page.mediabox	the page's `mediabox`
Page.number	page number
Page.parent	owning document object
Page.rect	rectangle of the page
Page.rotation_matrix	PDF only: get coordinates in rotated page space
Page.rotation	PDF only: page rotation
Page.transformation_matrix	PDF only: translate between PDF and MuPDF space
Page.xref	PDF only: page *xref*

Class API

`class Page`

`bound()`

Determine the rectangle of the page. Same as property *Page.rect*. 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*)

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.

- New in v1.16.0

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*) – choose one of "Note" (default), "Comment", "Help", "Insert", "Key", "NewParagraph", "Paragraph" as the visual symbol for the embodied text[4]. (New in v1.16.0)

Return type
Annot

Returns
the created annotation. Stroke color yellow = (1, 1, 0), no fill color support.

add_freetext_annot(*rect*, *text*, ***, *fontsize=11*, *fontname='helv'*, *text_color=0*, *fill_color=None*, *border_width=0*, *dashes=None*, *callout=None*, *line_end=PDF_ANNOT_LE_OPEN_ARROW*, *opacity=1*, *align=TEXT_ALIGN_LEFT*, *rotate=0*, *richtext=False*, *style=None*)

PDF only: Add text in a given rectangle. Optionally, the appearance of a "callout" shape can be requested by specifying two or three point-like objects – see below.

Parameters

- **rect** (*rect_like*) – the rectangle into which the text should be inserted. Text is automatically wrapped to a new line at box width. Text portions not fitting into the rectangle will be invisible without warning.

[4] You are generally free to choose any of the *Annotation Icons in MuPDF* you consider adequate.

- **text** (*str*) – the text. May contain any mixture of Latin, Greek, Cyrillic, Chinese, Japanese and Korean characters. If `richtext=True` (see below), the string is interpreted as HTML syntax. This adds a plethora of ways for attractive effects.
- **fontsize** (*float*) – the *fontsize*. Default is 11. Ignored if `richtext=True`.
- **fontname** (*str*) – The font name. Default is "Helv". Ignored if `richtext=True`, otherwise the following **restritions apply:**
  - Accepted alternatives are "Helv" (Helvetica), "Cour" (Courier), "TiRo" (Timnes-Roman), "ZaDb" (ZapfDingBats) and "Symb" (Symbol). The name may be abbreviated to the first two characters, like "Co" for "Cour", lower case accepted.
  - Bold or italic variants of the fonts are **not supported.**
- **text_color** (*list,tuple,float*) – the text color. Default is black. Ignored if `richtext=True`.
- **fill_color** (*list,tuple,float*) – the fill color. This is used for `rect` and the end point of the callout lines when applicable. Default is `None`.
- **border_color** (*list,tuple,float*) – This parameter **only has an effect** if `richtext=True`. Otherwise, `text_color` is used.
- **border_width** (*float*) – the width of border and `callout` lines. Default is 0 (no border), in which case callout lines may still appear with some hairline width, depending on the PDF viewer used. In any case, this value must be positive to see a border line.
- **dashes** (*list,tuple*) – a list of floats specifying how border and callout lines should be dashed. Default is `None`.
- **callout** (*list,tuple*) – a list / tuple of two or three *point_like* objects, which will be interpreted as end point [, knee point] and start point (in this sequence) of up to two line segments, converting this annotation into a call-out shape.
- **line_end** (*int*) – the line end symbol of the call-out line. It is drawn at the first point specified in the `callout` list. Default is an open arrow. For possible values see *Annotation Line Ending Styles*.
- **opacity** (*float*) – a float `0 <= opacity < 1` turning the annotation transparent. Default is no transparency.
- **align** (*int*) – text alignment, one of TEXT_ALIGN_LEFT, TEXT_ALIGN_CENTER, TEXT_ALIGN_RIGHT - justify is **not supported**. Ignored if `richtext=True`.
- **rotate** (*int*) – the text orientation. Accepted values are integer multiples of 90°. Invalid entries receive a rotation of 0.
- **richtext** (*bool*) – treat `text` as HTML syntax. This allows to achieve **bold**, *italic*, arbitrary text colors, font sizes, text alignment including justify and more - as far as the PDF subset of HTML and styling instructions supports this. This is similar to what happens in *Page.insert_htmlbox()*. The base library will for example pull in required fonts if it encounters characters not contained in the standard ones. Some parameters are ignored if this option is set, as mentioned above. Default is `False`.
- **style** (*str*) – supply optional HTML styling information in CSS syntax. Ignored if `richtext=False`.

Return type
Annot

Returns
the created annotation.

- Changed in v1.19.6: add border color parameter

`add_file_annot`(*point*, *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*) – choose one of "PushPin" (default), "Graph", "Paperclip", "Tag" as the visual symbol for the attached data[Page 304, 4]. (New in v1.16.0)

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.

Redactions

`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*)

PDF only: Add a redaction annotation. A redaction annotation identifies an area whose content should be removed from the document. Adding such an annotation is the first of two steps. It makes visible what will be removed in the subsequent step, *Page.apply_redactions()*.

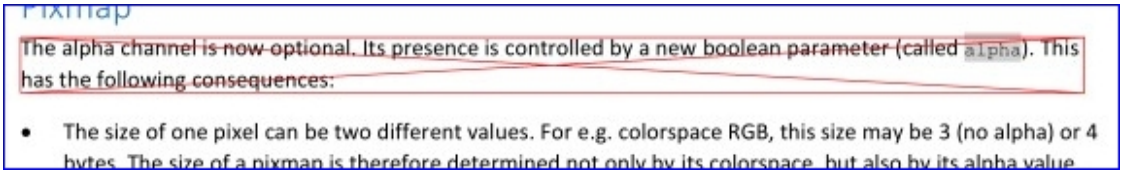
Parameters

- **quad** (*quad_like,rect_like*) – specifies the (rectangular) area to be removed which is always equal to the annotation rectangle. This may be a *rect_like* or *quad_like* object. If a quad is specified, then the enveloping rectangle is taken.
- **text** (*str*) – text to be placed in the rectangle after applying the redaction (and thus removing old content). (New in v1.16.12)
- **fontname** (*str*) – the font to use when `text` is given, otherwise ignored. Only CJK and the *PDF Base 14 Fonts* are supported. Apart from this, the same rules apply as for *Page.insert_textbox()* – which is what the method *Page.apply_redactions()* internally invokes.
- **fontsize** (*float*) – 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. (New in v1.16.12)
- **align** (*int*) – the horizontal alignment for the replacing text. See *insert_textbox()* for available values. The vertical alignment is (approximately) centered.
- **fill** (*sequence*) – 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. To suppress a fill color altogether, specify `False`. In this cases the rectangle remains transparent. (New in v1.16.12)

- **text_color** (*sequence*) – the color of the replacing text. Default is *black = (0, 0, 0)*. (New in v1.16.12)
- **cross_out** (*bool*) – add two diagonal lines to the annotation rectangle. (New in v1.17.2)

Return type
Annot

Returns
the created annotation. 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. (Changed in v1.17.2)



- New in v1.16.11

apply_redactions(*images=PDF_REDACT_IMAGE_PIXELS | 2*, *graphics=PDF_REDACT_LINE_ART_REMOVE_IF_COVERED | 1*, *text=PDF_REDACT_TEXT_REMOVE | 0*)

PDF only: Remove all **content** contained in any redaction rectangle on the page.

This method applies and then deletes all redactions from the page.

Parameters

- **images** (*int*) – How to redact overlapping images. The default `PDF_REDACT_IMAGE_PIXELS | 2` blanks out overlapping pixels. `PDF_REDACT_IMAGE_NONE | 0` ignores, and `PDF_REDACT_IMAGE_REMOVE | 1` completely removes images overlapping any redaction annotation. Option `PDF_REDACT_IMAGE_REMOVE_UNLESS_INVISIBLE | 3` only removes images that are actually visible.
- **graphics** (*int*) – How to redact overlapping vector graphics (also called "line-art" or "drawings"). The default `PDF_REDACT_LINE_ART_REMOVE_IF_COVERED | 1` removes any overlapping vector graphics. `PDF_REDACT_LINE_ART_NONE | 0` ignores, and `PDF_REDACT_LINE_ART_REMOVE_IF_TOUCHED | 2` removes graphics fully contained in a redaction annotation. When removing line-art, please be aware that **stroked** vector graphics (i.e. type "s" or "sf") have a **larger wrapping rectangle** than one might expect: first of all, at least 50% of the path's line width have to be added in each direction to truly include all of the drawing. If a so-called "miter limit" is provided (see page 121 of the PDF specification), the enlarging value is `miter * width / 2`. So, when letting everything default (width = 1, miter = 10), the redaction rectangle should be at least 5 points larger in every direction.
- **text** (*int*) – Whether to redact overlapping text. The default `PDF_REDACT_TEXT_REMOVE | 0` removes all characters whose boundary box overlaps any redaction rectangle. This complies with the original legal / data protection intentions of redaction annotations. Other use cases however may require to

keep text while redacting vector graphics or images. This can be achieved by setting `text=True|PDF_REDACT_TEXT_NONE | 1`. This does **not comply** with the data protection intentions of redaction annotations. **Do so at your own risk.**

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.
- The overlapping parts of **images** will be blanked-out for default option `PDF_REDACT_IMAGE_PIXELS` (changed in v1.18.0). Option 0 does not touch any images and 1 will remove any image with an overlap.
- 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*.

- 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.

- Changed in v1.23.27: added option for removing graphics as well.
- Changed in v1.24.2: added option `keep_text` to leave text untouched.

---

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:

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*) – the location(s) – rectangle(s) or quad(s) – to be marked. (Changed in v1.14.20) 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). **Set this parameter to** `None` if you want to use the following arguments (Changed in v1.16.14). And vice versa: if not `None`, the remaining parameters must be `None`.
- **start** (*point_like*) – start text marking at this point. Defaults to the top-left point of *clip*. Must be provided if `quads` is `None`. (New in v1.16.14)
- **stop** (*point_like*) – stop text marking at this point. Defaults to the bottom-right point of *clip*. Must be used if `quads` is `None`. (New in v1.16.14)
- **clip** (*rect_like*) – only consider text lines intersecting this area. Defaults to the page rectangle. Only use if `start` and `stop` are provided. (New in v1.16.14)

Return type

Annot or `None` (changed in v1.16.14).

Returns

the created annotation. If *quads* is an empty list, **no annotation** is created (changed in v1.16.14).

Note

You can use parameters *start*, *stop* and *clip* to highlight consecutive lines between the points *start* and *stop* (starting with v1.16.14). 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.

cluster_drawings(*clip=None*, *drawings=None*, *x_tolerance=3*, *y_tolerance=3*, *final_filter=True*)

Cluster vector graphics (synonyms are line-art or drawings) based on their geometrical vicinity. The method walks through the output of *Page.get_drawings()* and joins paths whose `path["rect"]` are closer to each other than some tolerance values (given in the arguments). The result is a list of rectangles that each wrap things like tables (with gridlines), pie charts, bar charts, etc.

Parameters

- **clip** (*rect_like*) – only consider paths inside this area. The default is the full page.
- **drawings** (*list*) – (optional) provide a previously generated output of *Page.get_drawings()*. If `None` the method will execute the method.
- **y_tolerance** (*float x_tolerance /*) – Assume vector graphics to be close enough neighbors for belonging to the same rectangle. Default is 3 points.
- **final_filter** (*bool*) – If `True` (default), the method will to remove rectangles having width or height smaller than the respective tolerance value. If `False` no such filtering is done.

find_tables(*clip=None*, *strategy=None*, *vertical_strategy=None*, *horizontal_strategy=None*, *vertical_lines=None*, *horizontal_lines=None*, *snap_tolerance=None*, *snap_x_tolerance=None*, *snap_y_tolerance=None*, *join_tolerance=None*, *join_x_tolerance=None*, *join_y_tolerance=None*, *edge_min_length=3*, *min_words_vertical=3*, *min_words_horizontal=1*, *intersection_tolerance=None*, *intersection_x_tolerance=None*, *intersection_y_tolerance=None*, *text_tolerance=None*, *text_x_tolerance=None*, *text_y_tolerance=None*, *add_lines=None*, *add_boxes=None*, *paths=None*, *use_layout=True*)

Find tables on the page and return an object with related information. Typically, the default values of the many parameters will be sufficient. Adjustments should ever only be needed in corner case situations.

Parameters

- **clip** (*rect_like*) – specify a region to consider within the page rectangle and ignore the rest. Default is the full page.
- **strategy** (*str*) – Request a **table detection** strategy. Valid values are "lines", "lines_strict" and "text".

  Default is **"lines"** which uses all vector graphics on the page to detect grid lines.

  Strategy **"lines_strict"** ignores borderless rectangle vector graphics. Sometimes single text pieces have background colors which may lead to false columns or lines. This strategy ignores them and can thus increase detection precision.

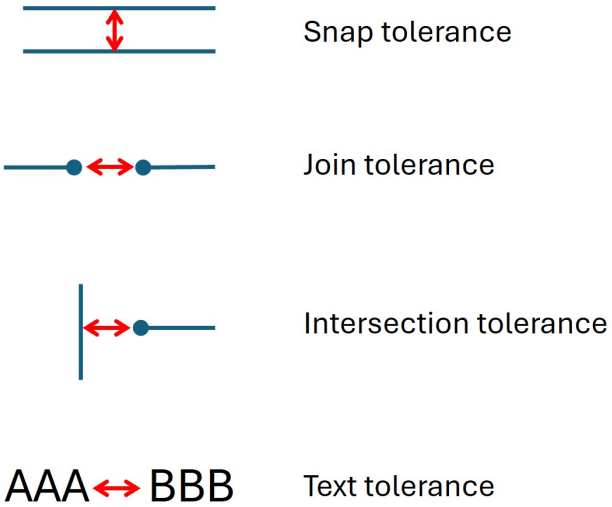
  If **"text"** is specified, text positions are used to generate "virtual" column and / or row boundaries. Use `min_words_*` to request the number of words for considering their coordinates.

Use parameters `vertical_strategy` and `horizontal_strategy` **instead** for a more fine-grained treatment of the dimensions.

- **horizontal_lines** (*sequence[floats]*) – y-coordinates of rows. If provided, there will be no attempt to identify additional table rows. This influences table detection.
- **vertical_lines** (*sequence[floats]*) – x-coordinates of columns. If provided, there will be no attempt to identify additional table columns. This influences table detection.
- **min_words_vertical** (*int*) – relevant for vertical strategy option "text": at least this many words must coincide to establish a **virtual column** boundary.
- **min_words_horizontal** (*int*) – relevant for horizontal strategy option "text": at least this many words must coincide to establish a **virtual row** boundary.
- **snap_tolerance** (*float*) – Any two horizontal lines whose y-values differ by no more than this value will be **snapped** into one. Accordingly for vertical lines. Default is 3. Separate values can be specified instead for the dimensions, using `snap_x_tolerance` and `snap_y_tolerance`.
- **join_tolerance** (*float*) – Any two lines will be **joined** to one if the end and the start points differ by no more than this value (in points). Default is 3. Instead of this value, separate values can be specified for the dimensions using `join_x_tolerance` and `join_y_tolerance`.
- **edge_min_length** (*float*) – Ignore a line if its length does not exceed this value (points). Default is 3.
- **intersection_tolerance** (*float*) – When combining lines into cell borders, orthogonal lines must be within this value (points) to be considered intersecting. Default is 3. Instead of this value, separate values can be specified for the dimensions using `intersection_x_tolerance` and `intersection_y_tolerance`.
- **text_tolerance** (*float*) – Characters will be combined into words only if their distance is no larger than this value (points). Default is 3. Instead of this value, separate values can be specified for the dimensions using `text_x_tolerance` and `text_y_tolerance`.
- **add_lines** (*tuple,list*) – Specify a list of "lines" (i.e. pairs of *point_like* objects) as **additional**, "virtual" vector graphics. These lines may help with table and / or cell detection and will not otherwise influence the detection strategy. Especially, in contrast to parameters `horizontal_lines` and `vertical_lines`, they will not prevent detecting rows or columns in other ways. These lines will be treated exactly like "real" vector graphics in terms of joining, snapping, intersecting, minimum length and containment in the `clip` rectangle. Similarly, lines not parallel to any of the coordinate axes will be ignored.
- **add_boxes** (*tuple,list*) – Specify a list of rectangles (*rect_like* objects) as **additional**, "virtual" vector graphics. These rectangles may help with table and / or cell detection and will not otherwise influence the detection strategy. Especially, in contrast to parameters `horizontal_lines` and `vertical_lines`, they will not prevent detecting rows or columns in other ways. These rectangles will be treated exactly like "real" vector graphics in terms of joining, snapping, intersecting, minimum length and containment in the `clip` rectangle.
- **paths** (*list*) – list of vector graphics in the format as returned be *Page.get_drawings()*. Using this parameter will prevent the method to extract vector

graphics itself. This is useful if the vector graphics are already available. This can save execution time significantly.

- **use_layout** (*bool*) – use layout analysis to gate line-based table candidates. Set to `False` for pure line-based detection.



Returns

a *TableFinder* object that has the following significant attributes:

- *cells*: a list of **all bboxes** on the page, that have been identified as table cells (across all tables). Each cell is a *rect_like* tuple `(x0, y0, x1, y1)` of coordinates or `None`.
- *tables*: a list of *Table* objects. This is `[]` if the page has no tables. Single tables can be found as items of this list. But the *TableFinder* object itself is also a sequence of its tables. This means that if `tabs` is a *TableFinder* object, then table "n" is delivered by `tabs.tables[n]` as well as by the shorter `tabs[n]`.
- The *Table* object has the following attributes:
  - `bbox`: the bounding box of the table as a tuple `(x0, y0, x1, y1)`.
  - `cells`: bounding boxes of the table's cells (list of tuples). A cell may also be `None`.
  - `extract()`: this method returns the text content of each table cell as a list of list of strings.
  - `to_markdown()`: this method returns the table as a **string in markdown format** (compatible to Github). Markdown viewers can render the string as a table. This output is optimized for **small token** sizes, which is especially beneficial for LLM/RAG feeds. Pandas DataFrames (see method *to_pandas()* below) offer an equivalent markdown table output which however is better readable for the human eye. Any line breaks (`\n`) in cells are replaced by HTML line breaks tags `<br>`.
  - *to_pandas()*: this method returns the table as a pandas DataFrame. DataFrames are very versatile objects allowing a plethora of table manipulation methods and outputs to almost 20 well-known formats, among them Excel files, CSV, JSON, markdown-formatted tables and more. `DataFrame.to_markdown()` generates a Github-compatible markdown format optimized for human readability. This method however requires the package tabulate to be installed in addition to pandas itself.

  - `header`: a *TableHeader* object containing header information of the table.
  - `col_count`: an integer containing the number of table columns.
  - `row_count`: an integer containing the number of table rows.
  - `rows`: a list of *TableRow* objects containing two attributes, `bbox` is the boundary box of the row, and *cells* is a list of table cells contained in this row.
- The *TableHeader* object has the following attributes:
  - `bbox`: the bounding box of the header.
  - *cells*: a list of bounding boxes containing the name of the respective column.
  - *names*: a list of strings containing the text of each of the cell bboxes. They represent the column names – which are used when exporting the table to pandas DataFrames, markdown, etc.
  - *external*: a bool indicating whether the header bbox is outside the table body (`True`) or not. Table headers are never identified by the *TableFinder* logic. Therefore, if *external* is true, then the header cells are not part of any cell identified by *TableFinder*. If `external == False`, then the first table row is the header.

Please have a look at these Jupyter notebooks, which cover standard situations like multiple tables on one page or joining table fragments across multiple pages.

Caution

The lifetime of the *TableFinder* object, as well as that of all its tables **equals the lifetime of the page**. If the page object is deleted or reassigned, all tables are no longer valid.

The only way to keep table content beyond the page's availability is to **extract it** via methods *Table.to_markdown()*, *Table.to_pandas()* or a copy of *Table.extract()* (e.g. `Table.extract()[:]`).

Note

Once a table has been extracted to a **Pandas DataFrame** with *to_pandas()* it is easy to convert to other file types with the **Pandas API**:

- table to Markdown, use to_markdown
- table to JSON, use: to_json
- table to Excel, use: to_excel
- table to CSV, use: to_csv
- table to HTML, use: to_html
- table to SQL, use: to_sql

- New in version 1.23.0
- Changed in version 1.23.19: new argument `add_lines`.

> **Important**
>
> There is also the pdf2docx extract tables method which is capable of table extraction if you prefer.

`add_stamp_annot`(*rect*, *stamp=0*)

PDF only: Add a "rubber stamp" annotation to e.g. indicate the document's intended use ("DRAFT", "CONFIDENTIAL", etc.). The parameter may be either an integer to select text from a predefined array of standard texts or an image.

Parameters

- **rect** (*rect_like*) – rectangle where to place the annotation.
- **stamp** (*multiple*) – The following options are available:
  - The id number (int) of the stamp text. For available stamps see *Stamp Annotation Icons*.
  - A string specifying an image file path.
  - A `bytes`, `bytearray` or `io.BytesIO` object for an image in memory.
  - A *Pixmap*.

1. **Text-based stamps**
   - *Annot.rect* is automatically calculated as the largest rectangle with an aspect ratio of `width:height = 3.8` that fits in the provided `rect`. Its position is vertically and horizontally centered.
   - The font chosen is "Times Bold" and the text will be upper case.
   - The appearance can be modified using *Annot.set_opacity()* and by setting the "stroke" color. By PDF specification, stamp annotations have no "fill" color.

2. **Image-based stamps**
   - The image is scaled to fit into the rectangle *Rect* such that the image's center and the center of *Rect* coincide. The aspect ratio of the image is preserved, so the image may not fill the entire rectangle. However, at least one of the given rectangle's width or height are fully covered.
   - The annotation can be modified via *Annot.set_opacity()*. This method therefore is a way to display images transparently even if no alpha channel is present.
   - Setting colors has no effect on image stamps.
   - Rotating image-based stamps **is not supported**. Setting the rotation may lead to unexpected results.

`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*)

- The removal will now include any bound 'Popup' or response annotations and related objects (changed in v1.16.6).

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*)

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.

(New in v1.18.4)

delete_link(*linkdict*)

PDF only: Delete the specified link from the page. The parameter must be an **original item** of *get_links()*, see *Description of get_links() Entries*. 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 *Description of get_links() Entries*.

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 *Description of get_links() Entries*. 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()

PDF only: Return the label for the page.

Return type
str

Returns
the label string like "vii" for Roman numbering or "" if not defined.

- New in v1.18.6

get_links()

Retrieves **all** links of a page.

Return type
list

Returns
A list of dictionaries. For a description of the dictionary entries, see *Description of get_links() Entries*. Always use this or the *Page.links()* method if you intend to make changes to the links of a page.

links(*kinds=None*)

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=(pymupdf.LINK_GOTO,)* will only return internal links.

Return type
generator

Returns
an entry of *Page.get_links()* for each iteration.

- New in v1.16.4

annots(*types=None*)

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=(pymupdf.PDF_ANNOT_FREETEXT, pymupdf.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]:
```

- New in v1.16.4

widgets(*types=None*)

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=(pymupdf.PDF_WIDGET_TYPE_TEXT,)` will only return 'Text' fields.

Return type

generator

Returns

a *Widget* for each iteration.

- New in v1.16.4

write_text(*rect=None*, *writers=None*, *overlay=True*, *color=None*, *opacity=None*, *keep_proportion=True*, *rotate=0*, *oc=0*)

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*) – the *xref* of an *OCG* or *OCMD*. (New in v1.18.4)

Note

Parameters *overlay, keep_proportion, rotate* and *oc* have the same meaning as in *Page.show_pdf_page()*.

- New in v1.16.18

insert_text(*point*, *text*, ***, *fontsize=11*, *fontname='helv'*, *fontfile=None*, *idx=0*, *color=None*, *fill=None*, *render_mode=0*, *miter_limit=1*, *border_width=0.05*, *encoding=TEXT_ENCODING_LATIN*, *rotate=0*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *overlay=True*, *oc=0*)

PDF only: Insert text lines starting at *point_like* `point`. See *Shape.insert_text()*.

- Changed in v1.18.4

insert_textbox(*rect*, *buffer*, ***, *align=TEXT_ALIGN_LEFT*, *border_width=1*, *color=None*, *encoding=TEXT_ENCODING_LATIN*, *expandtabs=8*, *fill=None*, *fill_opacity=1*, *fontfile=None*, *fontname='helv'*, *fontsize=11*, *lineheight=None*, *miter_limit=1*, *morph=None*, *oc=0*, *overlay=True*, *render_mode=0*, *rotate=0*, *set_simple=False*, *stroke_opacity=1*)

PDF only: Insert text into the specified *rect_like* *rect*.

Parameters

overlay – see *Shape.commit()*.

For other args, see *Shape.insert_textbox*.

- Changed in v1.18.4

insert_htmlbox(*rect*, *text*, ***, *css=None*, *scale_low=0*, *archive=None*, *rotate=0*, *oc=0*, *opacity=1*, *overlay=True*)

PDF only: Insert text into the specified rectangle. The method has similarities with methods *Page.insert_textbox()* and *TextWriter.fill_textbox()*, but is **much more powerful**. This is achieved by letting a *Story* object do all the required processing.

- Parameter `text` may be a string as in the other methods. But it will be **interpreted as HTML source** and may therefore also contain HTML language elements – including styling. The `css` parameter may be used to pass in additional styling instructions.
- Automatic line breaks are generated at word boundaries. The "soft hyphen" character `""` (or ``) can be used to cause hyphenation and thus may also cause line breaks. **Forced** line breaks however are only achievable via the HTML tag `<br>` - `\n` is ignored and will be treated like a space.
- With this method the following can be achieved:
  - Styling effects like bold, italic, text color, text alignment, font size or font switching.
  - The text may include arbitrary languages – **including right-to-left** languages.
  - Scripts like Devanagari and several others in Asia have a highly complex system of ligatures, where two or more unicodes together yield one glyph. The Story uses the software package HarfBuzz , to deal with these things and produce correct output.
  - One can also **include images** via HTML tag `<img>` – the Story will take care of the appropriate layout. This is an alternative option to insert images, compared to *Page.insert_image()*.
  - HTML tables (tag `<table>`) may be included in the text and will be handled appropriately.
  - Links are automatically generated when present.
- If content does not fit in the rectangle, the developer has two choices:
  - **either** only be informed about this (and accept a no-op, just like with the other textbox insertion methods),
  - **or** (`scale_low=0` - the default) scale down the content until it fits.

Parameters

- **rect** (*rect_like*) – rectangle on page to receive the text.
- **text** (*str*,Story) – the text to be written. Can contain a mixture of plain text and HTML tags with styling instructions. Alternatively, a *Story* object may be specified (in which case the internal Story generation step will be omitted). A Story must have been generated with all required styling and Archive information.
- **css** (*str*) – optional string containing additional CSS instructions. This parameter is ignored if `text` is a Story. See *CSS Support* for more.
- **scale_low** (*float*) – if necessary, scale down the content until it fits in the target rectangle. This sets the down scaling limit. Default is 0, no limit. A value of 1 means no down-scaling permitted. A value of e.g. 0.2 means maximum down-scaling by 80%.
- **archive** (Archive) – an Archive object that points to locations where to find images or non-standard fonts. If `text` refers to images or non-standard fonts, this parameter is required. This parameter is ignored if `text` is a Story.

- **rotate** (*int*) – one of the values 0, 90, 180, 270. Depending on this, text will be filled:
  - 0: top-left to bottom-right.
  - 90: bottom-left to top-right.
  - 180: bottom-right to top-left.
  - 270: top-right to bottom-left.

- **oc** (*int*) – the xref of an *OCG* / *OCMD* or 0. Please refer to *Page.show_pdf_page()* for details.
- **opacity** (*float*) – set the fill and stroke opacity of the content. Only values `0 <= opacity < 1` are considered.
- **overlay** (*bool*) – put the text in front of other content. Please refer to *Page.show_pdf_page()* for details.

Returns

A tuple of floats `(spare_height, scale)`.

- spare_height: The (positive) height of the remaining space in *Rect* below the text, or -1 if we failed to fit.
- scale: The scaling required; `0 < scale <= 1`. Will be `scale_low` if we failed to fit.

Please refer to examples in this section of the recipes: *How to Fill a Box with HTML Text*.

- New in v1.26.5:
  - do additional scaling to fit long words.
  - If we succeeded and scaled down, the returned `spare_height` is now generally positive instead of being fixed to zero, because the final rect's height is usually not an exact multiple of the font line height.
- New in v1.23.8: rebased-only.
- New in v1.23.9: *opacity* parameter.

Drawing Methods

draw_line(*p1*, *p2*, *color=(0,)*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

PDF only: Draw a line from *p1* to *p2* (*point_like* s). See *Shape.draw_line()*.

- Changed in v1.18.4

draw_zigzag(*p1*, *p2*, *breadth=2*, *color=(0,)*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

PDF only: Draw a zigzag line from *p1* to *p2* (`point_like` s). See `Shape.draw_zigzag()`.

- Changed in v1.18.4

draw_squiggle(*p1*, *p2*, *breadth=2*, *color=(0,)*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

PDF only: Draw a squiggly (wavy, undulated) line from *p1* to *p2* (`point_like` s). See `Shape.draw_squiggle()`.

- Changed in v1.18.4

draw_circle(*center*, *radius*, *color=(0,)*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

PDF only: Draw a circle around *center* (`point_like`) with a radius of *radius*. See `Shape.draw_circle()`.

- Changed in v1.18.4

draw_oval(*quad*, *color=(0,)*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

PDF only: Draw an oval (ellipse) within the given `rect_like` or `quad_like`. See `Shape.draw_oval()`.

- Changed in v1.18.4

draw_sector(*center*, *point*, *angle*, *color=(0,)*, *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*)

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

- Changed in v1.18.4

draw_polyline(*points*, *color=(0,)*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *closePath=False*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

PDF only: Draw several connected lines defined by a sequence of *point_like* s. See *Shape.draw_polyline()*.

- Changed in v1.18.4

draw_bezier(*p1*, *p2*, *p3*, *p4*, *color=(0,)*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *closePath=False*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

PDF only: Draw a cubic Bézier curve from *p1* to *p4* with the control points *p2* and *p3* (all are *point_like* s). See *Shape.draw_bezier()*.

- Changed in v1.18.4

draw_curve(*p1*, *p2*, *p3*, *color=(0,)*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *closePath=False*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

PDF only: This is a special case of *draw_bezier()*. See *Shape.draw_curve()*.

- Changed in v1.18.4

draw_rect(*rect*, *color=(0,)*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *radius=None*, *oc=0*)

PDF only: Draw a rectangle. See *Shape.draw_rect()*.

- Changed in v1.18.4
- Changed in v1.22.0: Added parameter *radius*.

draw_quad(*quad*, *color=(0,)*, *fill=None*, *width=1*, *dashes=None*, *lineCap=0*, *lineJoin=0*, *overlay=True*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

PDF only: Draw a quadrilateral. See *Shape.draw_quad()*.

- Changed in v1.18.4

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.

- **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".

Rytpe
int

Returns
the *xref* of the installed font.

Note

Built-in fonts will not lead to the inclusion of a font file. So the resulting PDF file will remain small. However, your PDF viewer software is responsible for generating an appropriate appearance – and there **exist** differences on whether or how each one of them does this. This is especially true for the CJK fonts. But also Symbol and ZapfDingbats are incorrectly handled in some cases. Following are the **Font Names** and their correspondingly installed **Base Font** names:

Base-14 Fonts[1]

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

CJK Fonts[2] (China, Japan, Korea)

Font Name	Installed Base Font	Comments
china-s	Heiti	simplified Chinese
china-ss	Song	simplified Chinese (serif)
china-t	Fangti	traditional Chinese
china-ts	Ming	traditional Chinese (serif)
japan	Gothic	Japanese
japan-s	Mincho	Japanese (serif)
korea	Dotum	Korean
korea-s	Batang	Korean (serif)

insert_image(*rect*, *, *alpha=-1*, *filename=None*, *height=0*, *keep_proportion=True*, *mask=None*, *oc=0*, *overlay=True*, *pixmap=None*, *rotate=0*, *stream=None*, *width=0*, *xref=0*)

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.

Parameters

- **rect** (*rect_like*) – where to put the image. Must be finite and not empty.
- **alpha** (*int*) – deprecated and ignored.
- **filename** (*str*) – name of an image file (all formats supported by MuPDF – see *Supported Input Image Formats*).
- **height** (*int*)
- **keep_proportion** (*bool*) – maintain the aspect ratio of the image.
- **mask** (*bytes,bytearray,io.BytesIO*) – 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.
- **oc** (*int*) – (*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.
- **overlay** – see *Common Parameters*.
- **pixmap** (*Pixmap*) – a pixmap containing the image.
- **rotate** (*int*) – rotate the image. Must be an integer multiple of 90 degrees. Positive values rotate anti-clockwise. 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.
- **stream** (*bytes,bytearray,io.BytesIO*) – image in memory (all formats supported by MuPDF – see *Supported Input Image Formats*).
- **width** (*int*)

[1] If your existing code already uses the installed base name as a font reference (as it was supported by PyMuPDF versions earlier than 1.14), this will continue to work.

[3] Not all PDF readers display these fonts at all. Some others do, but use a wrong character spacing, etc.

[2] Not all PDF reader software (including internet browsers and office software) display all of these fonts. And if they do, the difference between the **serifed** and the **non-serifed** version may hardly be noticeable. But serifed and non-serifed versions lead to different installed base fonts, thus providing an option to be displayable with your specific PDF viewer.

- **xref** (*int*) – 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.

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 = pymupdf.open(...)
>>> rect = pymupdf.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 the 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 are ways to control the image size – even if transparency comes into play. Have a look at *How to Add Images to a PDF Page*.
5. The image is stored in the PDF at its original quality level. This may be much better than what you 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.

- Changed in v1.14.1: By default, the image keeps its aspect ratio.
- Changed in v1.14.11: Added args `keep_proportion`, `rotate`.
- 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.

- Added support for *stream* as `io.BytesIO`.

- Changed in v1.17.6: Insertion rectangle no longer needs to have a non-empty intersection with the page's *Page.cropbox*[5].
- Changed in v1.18.1: Added `mask` arg.
- Changed in v1.18.3: Added `oc` arg.
- Changed in v1.18.13:
  - Allow providing the image as the xref of an existing one.
  - Added *xref* arg.
  - Return *xref* of stored image.
- Changed in v1.19.3: deprecate and ignore *alpha* arg.

replace_image(*xref*, *filename=None*, *pixmap=None*, *stream=None*)

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.

- New in v1.21.0

delete_image(*xref*)

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.

If you inspect / extract a page's images by methods like *Page.get_images()*, *Page.get_image_info()* or *Page.get_text()*, the replacing "dummy" image will be detected like so `(45, 47, 1, 1, 8,`

[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.

`'DeviceGray', '', 'Im1', 'FlateDecode'` and also seem to "cover" the same boundary box on the page.

- New in v1.21.0

get_text(*option*, *, *clip=None*, *flags=None*, *textpage=None*, *sort=False*, *delimiters=None*)

Retrieves the content of a page in a variety of formats. Depending on the `flags` value, this may include text, images and several other object types. The method is a wrapper for multiple *TextPage* methods by choosing the output option `opt` as follows:

- "text" – *TextPage.extractTEXT()*, default. Always includes **text only.**
- "blocks" – *TextPage.extractBLOCKS()*. Includes text and **may** include image meta information.
- "words" – *TextPage.extractWORDS()*. Always includes **text only.**
- "html" – *TextPage.extractHTML()*. May include text and images.
- "xhtml" – *TextPage.extractXHTML()*. May include text and images.
- "xml" – *TextPage.extractXML()*. Always includes **text only.**
- "dict" – *TextPage.extractDICT()*. May include text and images.
- "json" – *TextPage.extractJSON()*. May include text and images.
- "rawdict" – *TextPage.extractRAWDICT()*. May include text and images.
- "rawjson" – *TextPage.extractRAWJSON()*. May include text and images.

Parameters

- **opt** (*str*) – A string indicating the requested format, one of the above. A mixture of upper and lower case is supported. If misspelled, option "text" is silently assumed.
- **clip** (*rect-like*) – restrict the extraction to this rectangle. If `None` (default), the visible part of the page is taken. Any content (text, images) that is **not fully contained** in `clip` will be completely omitted. To avoid clipping altogether use `clip=pymupdf.INFINITE_RECT()`. Only then the extraction will contain all items. This parameter has **no effect** on options "html", "xhtml" and "xml".
- **flags** (*int*) – indicator bits to control whether to include images or how text should be handled with respect to white spaces and `ligatures`. See *Font Properties* for available indicators and *Text Extraction Flags Defaults* for default settings. (New in v1.16.2)
- **textpage** – 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*) – 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. For options "blocks", "dict", "json", "rawdict", "rawjson", sorting happens by coordinates `(y1, x0)` of the respective block bbox. For options "words" and "text", the text lines are completely re-synthesized to follow the reading sequence and appearance in the document – which even establishes the original layout to some extent.

- **delimiters** (*str*) – use these characters as *additional* word separators with the "words" output option (ignored otherwise). By default, all white spaces (including non-breaking space `0xA0`) indicate start and end of a word. Now you can specify more characters causing this. For instance, the default will return `"john.doe@outlook.com"` as **one** word. If you specify `delimiters="@."` then the **four** words `"john"`, `"doe"`, `"outlook"`, `"com"` will be returned. Other possible uses include ignoring punctuation characters `delimiters=string.punctuation`. The "word" strings will not contain any delimiting character. (New in v1.23.5)

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* 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: 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.

- 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.
- Changed in v1.23.5: added `delimiters` parameter
- Changed in v1.24.11: changed the effect of `sort_True` for "text" and "words" to closely follow natural reading sequence.

`get_textbox`(*rect*, *textpage=None*)

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. It is based on dedicated code (changed in v1.19.0). A typical use is checking the result of *Page.search_for()*:

```
>>> rl = page.search_for("currency:")
>>> page.get_textbox(rl[0])
```

(continues on next page)

(continued from previous page)

```
'Currency:'
>>>
```

- New in v1.17.7
- Changed in v1.19.0: add *TextPage* parameter

get_textpage(*clip=None*, *flags=3*)

Create a *TextPage* for the page.

Parameters

- **flags** (*int*) – indicator bits controlling the content available for subsequent text extractions and searches – see the parameter of *Page.get_text()*.
- **clip** (*rect-like*) – restrict extracted text to this area. (New in v1.17.7)

Returns

TextPage

- New in v1.16.5
- Changed in v1.17.7: introduced `clip` parameter.

get_textpage_ocr(*flags=3*, *language='eng'*, *dpi=72*, *full=False*, *tessdata=None*)

Optical Character Recognition (**OCR**) technology can be used to extract text data for pages where text is in raster image or vector graphic format. Use this method to **OCR** a page for subsequent text extraction.

This method returns a *TextPage* for the page that includes OCRed text. MuPDF will invoke Tesseract-OCR if this method is used.

Parameters

- **flags** (*int*) – indicator bits controlling the content available for subsequent test 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 only page areas that contain no legible text.
- **tessdata** (*str*) – The name of Tesseract's language support folder `tessdata`. If omitted, the name is determined using function *get_tessdata()*.

> **Note**
>
> This method does **not** support a clip parameter – OCR (full or partial) will always happen for the complete page rectangle.

Returns
a *TextPage*. Execution may be significantly longer than `Page.get_textpage()`.

For `full=True` OCR, **all text** will have the font "GlyphLessFont" from Tesseract. In case of partial OCR (`full=False`), legible normal text will keep its properties, and only recognized text will have the GlyphLessFont.

Recognized / OCR text will follow (legible) normal text for partial OCR and will thus not be in reading order. Establishing reading order is – as always – your responsibility.

> **Note**
>
> Text extraction results, including any OCR, are stored in the returned *TextPage*. To access them, you must use the `textpage` parameter in all subsequent text extraction and search methods.
>
> This Jupyter notebook walks through an example for using OCR textpages.

- New in v.1.19.0
- Changed in v1.19.1: support full and partial OCRing a page.
- changed in v1.27.2: For partial OCR, **all** page areas outside legible text are now OCRed, not just those within images. This means that OCR will now also be performed for vector graphics, and for text containing illegible characters.

`get_drawings`(*extended=False*)

Return the vector graphics of the page. These are instructions which draw lines, rectangles, quadruples or curves, including properties like colors, transparency, line width and dashing, etc. Alternative terms are "line art" and "drawings".

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, so we adopted that name here, but the method **works for all document types**.

The path dictionary for fill, stroke and fill-stroke paths has been designed to be compatible with class *Shape*. There are the following keys:

Key	Value
closePath	Same as the parameter in *Shape*.
color	Stroke color (see *Shape*).
dashes	Dashed line specification (see *Shape*).
even_odd	Fill colors of area overlaps – same as the parameter in *Shape*.
fill	Fill color (see *Shape*).
items	List of draw commands: lines, rectangles, quads or curves.
lineCap	Number 3-tuple, use its max value on output with *Shape*.
lineJoin	Same as the parameter in *Shape*.
fill_opacity	fill color transparency (see *Shape*). (New in v1.18.17)
stroke_opacity	stroke color transparency (see *Shape*). (New in v1.18.17)
rect	Page area covered by this path. Information only.
layer	name of applicable Optional Content Group. (New in v1.22.0)
level	the hierarchy level if `extended=True`. (New in v1.22.0)
seqno	command number when building page appearance. (New in v1.19.0)
type	type of this path. (New in v1.18.17)
width	Stroke line width. (see *Shape*).

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()`*. (Changed in v1.18.17)

For paths other than groups or clips, 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, not applicable ones are present with a value of `None`: `"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*. Multiple rectangles within the same path are now detected (changed in v1.18.17). Integer `orientation` is 1 resp. -1 indicating whether the enclosed area is rotated left (1 = anti-clockwise), or resp. right[7] (changed in v1.19.2).
- `("qu", quad)` - a *Quad*. 3 or 4 consecutive lines are detected to actually represent a *Quad* (changed in v1.19.2:). (New in v1.18.17)

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 *How to Extract Drawings*.

Specifying `extended=True` significantly alters the output. Most importantly, new dictionary types are present: "clip" and "group". All paths will now be organized in a hierarchic structure which is encoded by the new integer key "level", the hierarchy level. Each group or clip establishes a new hierarchy, which applies to all subsequent paths having a *larger* level value. (New in v1.22.0)

[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.

Any path with a smaller level value than its predecessor will end the scope of (at least) the preceding hierarchy level. A "clip" path with the same level as the preceding clip will end the scope of that clip. Same is true for groups. This is best explained by an example:

```
+------+------+--------+------+--------+
| line | lvl0 | lvl1   | lvl2 |  lvl3  |
+------+------+--------+------+--------+
|  0   | clip |        |      |        |
|  1   |      | fill   |      |        |
|  2   |      | group  |      |        |
|  3   |      |        | clip |        |
|  4   |      |        |      | stroke |
|  5   |      |        | fill |        |  ends scope of clip in line 3
|  6   |      | stroke |      |        |  ends scope of group in line 2
|  7   |      | clip   |      |        |
|  8   | fill |        |      |        |  ends scope of line 0
+------+------+--------+------+--------+
```

The clip in line 0 applies to line including line 7. Group in line 2 applies to lines 3 to 5, clip in line 3 only applies to line 4.

"stroke" in line 4 is under control of "group" in line 2 and "clip" in line 3 (which in turn is a subset of line 0 clip).

- **"clip"** dictionary. Its values (most importantly "scissor") remain valid / apply as long as following dictionaries have a **larger "level"** value.

Key	Value
closePath	Same as in "stroke" or "fill" dictionaries
even_odd	Same as in "stroke" or "fill" dictionaries
items	Same as in "stroke" or "fill" dictionaries
rect	Same as in "stroke" or "fill" dictionaries
layer	Same as in "stroke" or "fill" dictionaries
level	Same as in "stroke" or "fill" dictionaries
scissor	the clip rectangle
type	"clip"

- "group" dictionary. Its values remain valid (apply) as long as following dictionaries have a **larger "level"** value. Any dictionary with an equal or lower level end this group.

Key	Value
rect	Same as in "stroke" or "fill" dictionaries
layer	Same as in "stroke" or "fill" dictionaries
level	Same as in "stroke" or "fill" dictionaries
isolated	(bool) Whether this group is isolated
knockout	(bool) Whether this is a "Knockout Group"
blendmode	Name of the BlendMode, default is "Normal"
opacity	Float value in range [0, 1].
type	"group"

> **Note**
>
> The method is based on the output of *Page.get_cdrawings()* – which is much faster, but requires somewhat more attention processing its output.

- 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 colorspaces.
- Changed in v1.19.2: add an indicator for the *"orientation"* of the area covered by an "re" item.
- Changed in v1.22.0: add new key `"layer"` which contains the name of the Optional Content Group of the path (or `None`).
- Changed in v1.22.0: add parameter `extended` to also return clipping and group paths.

get_cdrawings(*extended=False*)

Extract the vector graphics on the page. Apart from following technical differences, functionally equivalent to *Page.get_drawings()*, but much faster:

- 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.

If performance is a concern, 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.

- 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.
- Changed in v1.22.0: added new key `"layer"` which contains the name of the Optional Content Group of the path (or `None`).
- Changed in v1.22.0: added parameter `extended` to also return clipping paths.

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*)

Return a list of meta information dictionaries for all images displayed by the page. This works for all document types.

Parameters

- **hashes** (*bool*) – 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. (New in v1.18.13)
- **xrefs** (*bool*) – **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 its xref is undetectable for some reason. 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. (New in v1.18.13)

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"*. The dictionary layout is similar to that of image blocks in `page.get_text("dict")`.

In contrast to images included in *Page.get_text()*, image **binary content** is not loaded by this method, which drastically reduces memory usage. Another difference is that image detection is not restricted to the visible part of the page or any `clip` parameter: method *Page.get_text()* will only extract images **fully contained** in the provided `clip`.

Key	Value
number	block number (`int`)
bbox	image bbox on page, *rect_like*
width	original image width (`int`)
height	original image height (`int`)
cs-name	colorspace name (`str`)
colorspace	colorspace.n (`int`)
xres	resolution in x-direction (`int`)[10]
yres	resolution in y-direction (`int`)[10]
bpc	bits per component (`int`)
size	storage occupied by image (`int`)
digest	MD5 hashcode (`bytes`), if `hashes` is true
xref	image *xref* or 0, if *xrefs* is true
transform	matrix transforming image rect to bbox, *matrix_like*
has-mask	whether the image is transparent and has a mask (`bool`)

Multiple occurrences of the same image are always reported. You can detect duplicates by comparing their *digest* values.

- New in v1.18.11
- Changed in v1.18.13: added image MD5 hashcode computation and *xref* search.

[10] This value is always 96, the default of the PDF interpreter. It **does not reflect** the resolution of the image itself. If you need the image's resolution, use the *Pixmap.xres()* and *Pixmap.yres()* attributes of the *Pixmap* created from the image binary.

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*)

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.

New in v1.18.13

get_image_bbox(*item*, *transform=False*)

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*) – return the matrix used to transform the image rectangle to the bbox on the page (new in v1.18.11). 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 == pymupdf.Rect(0, 0, 1, 1)` is true, lookup details here: *Image Transformation Matrix*.

- Changed in v1.18.11: return image transformation matrix

`get_svg_image`(*matrix=pymupdf.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`*) – – 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. (New in v1.17.5)

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=pymupdf.Identity*, *dpi=None*, *colorspace=pymupdf.csRGB*, *clip=None*, *alpha=False*, *annots=True*)

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

All parameters are *keyword-only.*

Parameters

- **matrix** (*matrix_like*) – default is *Identity*.
- **dpi** (*int*) – desired resolution in x and y direction. If not `None`, the `"matrix"` parameter is ignored. (New in v1.19.2)
- **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 version 1.16.0)* whether to also render annotations or to suppress them. You can create pixmaps for annotations separately.

Return type
Pixmap

Returns
Pixmap of the page. For fine-controlling the generated image, the by far most important parameter is **matrix**. E.g. you can increase or decrease the image resolution by using **Matrix(xzoom, yzoom)**. If zoom > 1, you will get a higher resolution: zoom=2 will double the number of pixels in that direction and thus generate a 2 times larger image. Non-positive values will flip horizontally, resp. vertically. Similarly, matrices also let you rotate or shear, and you can combine effects via e.g. matrix multiplication. See the *Matrix* section to learn more.

Note

- The pixmap will have *"premultiplied"* pixels if `alpha=True`. To learn about some background, e.g. look for "Premultiplied alpha" here.
- 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 pymupdf
In [2]: doc=pymupdf.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]:
```

- Changed in v1.19.2: added support of parameter dpi.

`annot_names()`

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

- New in v1.16.10

`annot_xrefs()`

PDF only: return a list of the *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*.

- New in v1.17.1

`load_annot(ident)`

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.

- New in v1.17.1

load_widget(*xref*)

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.

- New in v1.19.6

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.

recolor(*components=1*)

PDF only: Change the colorspace components of all objects on page.

Parameters
components (*int*) – The desired count of color components. Must be one of 1, 3 or 4, which results in color spaces DeviceGray, DeviceRGB or DeviceCMYK respectively. The method affects text, images and vector graphics. For instance, with the default value 1, a page will be converted to grayscale. If a page is already grayscale, the method will not cause visible changes – independent of the value of components.

These changes are **permanent** and cannot be reverted.

clip_to_rect(*rect*)

PDF only: Permanently remove page content outside the given rectangle. This is similar to *Page.set_cropbox()*, but the page's rectangle will not be changed, only the content outside the rectangle will be removed.

Parameters
rect (*rect_like*) – The rectangle to clip to. Must be finite and its intersection with the page must not be empty.

The method works best for text: All text on the page will be removed (decided by single character) that has no intersection with the rectangle. For vector graphics, the method will remove all paths that have no intersection with the rectangle. For images, the method will remove all images that have no intersection with the rectangle. Vectors and images **having** an intersection with the rectangle, will be kept in their entirety.

The method roughly has the same effect as if four redactions had been applied that cover the rectangle's outside.

- New in v1.26.4.

remove_rotation()

PDF only: Set page rotation to 0 while maintaining appearance and page content.

Returns

The inverted matrix used to achieve this change. If the page was not rotated (rotation 0), *Identity* is returned. The method automatically recomputes the rectangles of any annotations, links and widgets present on the page.

This method may come in handy when e.g. used with *Page.show_pdf_page()*.

show_pdf_page(*rect*, *docsrc*, *pno=0*, *keep_proportion=True*, *overlay=True*, *oc=0*, *rotate=0*, *clip=None*)

PDF only: Display a page of another PDF. This is similar to *Page.insert_image()* but the source page will appear like a copy of itself and will not be rasterized. 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 combine.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.py, which puts an SVG-based logo on each page.

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 $-\infty$ < pno < 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*) – (*xref*) make visibility dependent on this *OCG* / *OCMD* (which must be defined in the target PDF)[9]. (New in v1.18.3)
- **rotate** (*float*) – show the source rectangle rotated by some angle. Any angle is supported (changed in v1.14.11). (New in v1.14.10)

[9] Objects inside the source page, like images, text or drawings, are never aware of whether their owning page now is under OC control inside the target PDF. If source page objects are OC-controlled in the source PDF, then this will not be retained on the target: they will become unconditionally visible.

- **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.

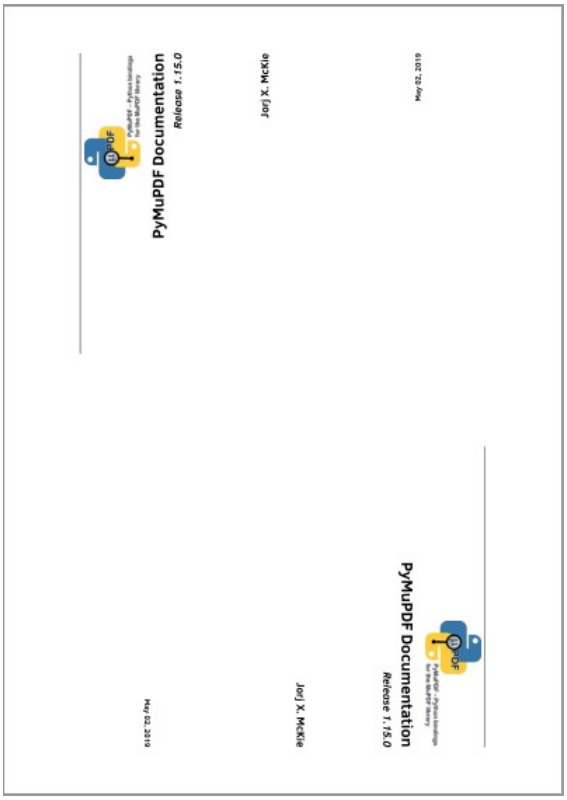
Note

In contrast to method *Document.insert_pdf()*, this method does not copy annotations, widgets or links, so these objects are not included in the target[6]. But all its **other resources (text, images, fonts, etc.)** will be imported into the current PDF. They will therefore appear in text extractions and in *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 = pymupdf.open()  # new empty PDF
>>> page=doc.new_page()  # new page in A4 format
>>>
>>> # upper half page
>>> r1 = pymupdf.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 = pymupdf.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")
```

[6] If you need to also see annotations or fields in the target page, you can convert the source PDF using *Document.bake()*. 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.



- 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`.

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=None*, *quads=False*, *flags=TEXT_DEHYPHENATE | TEXT_PRESERVE_WHITESPACE | TEXT_PRESERVE_LIGATURES | TEXT_MEDIABOX_CLIP*, *textpage=None*)

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*) – only search within this area. (New in v1.18.2)

- **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 hyphenation[8] is detected.
- **textpage** – 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. (New in v1.19.0)

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".
- 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

[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.

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")`.

- Changed in v1.18.2: added `clip` parameter. Remove `hit_max` parameter. Add default "dehyphenate".
- Changed in v1.19.0: added *TextPage* parameter.

set_mediabox(*r*)

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.

- New in v1.16.13
- Changed in v1.19.4: remove all other rectangle definitions.

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
pymupdf.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> page.cropbox  # cropbox and mediabox still equal
pymupdf.Rect(0.0, 0.0, 595.0, 842.0)
>>>
>>> # now set cropbox to a part of the page
>>> page.set_cropbox(pymupdf.Rect(100, 100, 400, 400))
>>> # this will also change the "rect" property:
>>> page.rect
pymupdf.Rect(0.0, 0.0, 300.0, 300.0)
>>>
>>> # but mediabox remains unaffected
>>> page.mediabox
pymupdf.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*)

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()`.

- New in v1.19.4

rotation

Contains the rotation of the page in degrees (always 0 for non-PDF types). This is a copy of the value in the PDF file. The PDF documentation says:

> *"The number of degrees by which the page should be rotated clockwise when displayed or printed. The value must be a multiple of 90. Default value: 0."*
>
> In PyMuPDF, we make sure that this attribute is always one of 0, 90, 180 or 270.
>
> **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 = pymupdf.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

27.14.2 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 *pymupdf.Point*, specifying the destination location on the provided page, default is *pymupdf.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 *Notes on Supporting Links*.

27.14.3 Notes on Supporting Links

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

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.

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": pymupdf.LINK_GOTO, "page": pno, "from": bbox, "to": point}`.
4. Execute `page.insert_link(d)`.

27.14.4 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
Document.get_page_fonts()	*Page.get_fonts()*
Document.get_page_images()	*Page.get_images()*
Document.get_page_pixmap()	*Page.get_pixmap()*
Document.get_page_text()	*Page.get_text()*
Document.search_page_for()	*Page.search_for()*

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)`.

When calling the *Document* equivalent methods then the page number is sent through as a parameter, e.g.:

`Document.get_page_images(pno)` or `Document.get_page_text(pno)`

Tip

The page number parameter, `pno`, is a 0-based integer $-\infty$ < `pno` < `page_count`.

27.14.5 Tables and Related Classes

The *TableFinder* class is returned by *Page.find_tables()* and has related classes as follows:

class TableFinder

An object always returned by *Page.find_tables()*. Attributes of interest:

tables

A list of *Table* objects, each of which represents a table found on the page. An empty list if no tables are found.

page

A reference to the *Page* object.

Type
Page

class Table

An object representing a table found on the page.

page

A back-reference to the owning page.

Type
Page

cells

An array of *Rect* objects for each cell in the table.

Type
list

header

A *TableHeader* object.

Type
TableHeader

bbox

The bounding box of all cells of the table header.

Type
Rect

row_count

Number of rows in the table.

Type
int

col_count

Number of columns in the table.

Type
int

rows

An array of *TableRow* objects for each row in the table.

Type
list

extract()

Extracts table cell text data into a list.

Type
list

to_markdown(*clean=False*, *fill_empty=True*)

Extracts table data into Markdown text format.

Parameters

- **clean** (*bool*) – If `True` then markdown syntax is removed from cell content.
- **fill_empty** (*bool*) – If `True` then cell content `None` is replaced by the values above (columns) or left (rows) in an effort to approximate row and columns spans.

Type
string

`to_pandas()`

Return a pandas DataFrame DataFrame version of the table.

Type
pandas DataFrame

`class TableHeader`

Dedicated class for table headers.

`bbox`

The bounding box of the union of cells belonging to the table header, given as a tuple (x0, y0, x1, y1). This rectangle contains all table header cells.

Type
Rect

`cells`

A list of tuples for each bbox of a column header.

Type
list

`names`

A list of strings with column header text.

Type
list

`external`

A boolean indicating whether the header is outside the table cells.

Type
`bool`

`class TableRow`

Dedicated class for table rows.

This documentation covers all versions up to 1.28.2.

27.15 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
`Pixmap.clear_with()`	clear parts of the pixmap
`Pixmap.color_count()`	determine used colors
`Pixmap.color_topusage()`	determine share of most used color
`Pixmap.copy()`	copy parts of another pixmap
`Pixmap.flip_rotate()`	rotate or flip the pixmap
`Pixmap.gamma_with()`	apply a gamma factor to the pixmap
`Pixmap.invert_irect()`	invert the pixels of a given area
`Pixmap.pdfocr_save()`	save the pixmap as an OCRed 1-page PDF
`Pixmap.pdfocr_tobytes()`	save the pixmap as an OCRed 1-page PDF
`Pixmap.pil_image()`	create a Pillow Image
`Pixmap.pil_save()`	save as a Pillow Image
`Pixmap.pil_tobytes()`	write to `bytes` as a Pillow Image
`Pixmap.pixel()`	return the value of a pixel
`Pixmap.save()`	save the pixmap in a variety of formats
`Pixmap.set_alpha()`	set alpha values
`Pixmap.set_dpi()`	set the image resolution
`Pixmap.set_origin()`	set pixmap x,y values
`Pixmap.set_pixel()`	set color and alpha of a pixel
`Pixmap.set_rect()`	set color and alpha of all pixels in a rectangle
`Pixmap.shrink()`	reduce size keeping proportions
`Pixmap.tint_with()`	tint the pixmap
`Pixmap.tobytes()`	return a memory area in a variety of formats
`Pixmap.warp()`	return a pixmap made from a quad inside
`Pixmap.alpha`	transparency indicator
`Pixmap.colorspace`	pixmap's *Colorspace*
`Pixmap.digest`	MD5 hashcode of the pixmap
`Pixmap.height`	pixmap height
`Pixmap.interpolate`	interpolation method indicator
`Pixmap.is_monochrome`	check if only black and white occur
`Pixmap.is_unicolor`	check if only one color occurs
`Pixmap.irect`	*IRect* of the pixmap
`Pixmap.n`	bytes per pixel
`Pixmap.samples_mv`	`memoryview` of pixel area
`Pixmap.samples_ptr`	Python pointer to pixel area

continues on next page

Table 4 – continued from previous page

Method / Attribute	Short Description
Pixmap.samples	bytes copy of pixel area
Pixmap.size	pixmap's total length
Pixmap.stride	size of one image row
Pixmap.width	pixmap width
Pixmap.x	X-coordinate of top-left corner
Pixmap.xres	resolution in X-direction
Pixmap.y	Y-coordinate of top-left corner
Pixmap.yres	resolution in Y-direction

Class API

class Pixmap

__init__(*self*, *colorspace*, *irect*, *alpha=False*)

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 graysale 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 = pymupdf.Pixmap(pix, 0)    # extract the RGB part (drop alpha)
>>> pixalpha = pymupdf.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 pimap properties are set by the image. Have a look at extract-from-pages.py and extract-from-xref.py to see how this can be used to recover all of a PDF's images.

Parameters

- **doc** (*Document*) – an opened 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(*black*, *white*)

Colorize a pixmap by replacing black and / or white with colors given as **sRGB integer** values. Only colorspaces *CS_GRAY* and *CS_RGB* are supported, others are ignored with a warning.

If the colorspace is *CS_GRAY*, the average *(red + green + blue)/3* will be taken. The pixmap will be changed in place.

Parameters

- **black** (*int*) – replace black with this value. Specifying 0x000000 makes no changes.
- **white** (*int*) – replace white with this value. Specifying 0xFFFFFF makes no changes.

Examples:

- `tint_with(0x000000, 0xFFFFFF)` is a no-op.
- `tint_with(0x00FF00, 0xFFFFFF)` changes black to green, leaves white intact.
- `tint_with(0xFF0000, 0x0000FF)` changes black to red and white to blue.

gamma_with(*gamma*)

Apply a gamma factor to a pixmap, i.e. lighten or darken it. Pixmaps with colorspace `None` are ignored with a warning.

Parameters

gamma (*float*) – *gamma* = *1.0* does nothing, *gamma* < *1.0* lightens, *gamma* > *1.0* darkens the image.

flip_rotate(*mode*)

Rotate the pixmap by a multiple of 90 degrees or flip it horizontally or vertically.

Parameters

mode (*int*) – `mode` is an integer to specify the desired action. Also see *Pixmap Transformations*. The following values are supported:

- 1: Pixmap.ROTATE_90 - rotate 90 degrees clockwise
- 2: Pixmap.ROTATE_270 - rotate 270 degrees clockwise
- 3: Pixmap.ROTATE_180 - rotate 180 degrees clockwise
- 4: Pixmap.FLIP_LEFT_RIGHT - flip horizontally
- 5: Pixmap.FLIP_TOP_BOTTOM - flip vertically

Returns

A new pixmap with the requested rotation or flip applied. The original pixmap is not changed. If an unsupported mode value is provided, a warning is issued and the original pixmap is returned.

shrink(*n*)

Shrink the pixmap by dividing both, its width and height by 2:sup:`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* colorspaces, but they currently **must** have the same alpha property[2]. 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

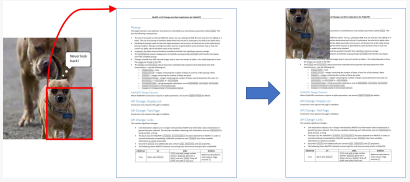
- **source** (*Pixmap*) – source pixmap.
- **irect** (*irect_like*) – The area to be copied.

[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*, *jpg_quality=95*)

- Changed in v1.22.0: Added **direct support of JPEG** images. Image quality can be controlled via parameter "jpg_quality".

Save pixmap as an image file. Depending on the output chosen, only some or all colorspaces are supported and different file extensions can be chosen. Please see the table below.

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 desired image format. The default is the filename's extension. If both, this value and the file extension are unsupported, an exception is raised. For possible values see *Supported Output Image Formats*.
- **jpg_quality** (*int*) – The desired image quality, default 95. Only applies to JPEG images, else ignored. This parameter trades quality against file size. A value of 98 is close to lossless. Higher values should not lead to better quality.

Raises

ValueError – For unsupported image formats.

tobytes(*output='png'*, *jpg_quality=95*)

- New in version 1.14.5: Return the pixmap as a *bytes* memory object of the specified format – similar to *save()*.
- Changed in v1.22.0: Added **direct JPEG support**. Image quality can be influenced via new parameter "jpg_quality".

Parameters

- **output** (*str*) – The desired image format. The default is "png". For possible values see *Supported Output Image Formats*.
- **jpg_quality** (*int*) – The desired image quality, default 95. Only applies to JPEG images, else ignored. This parameter trades quality against file size. A value of 98 is close to lossless. Higher values should not lead to better quality.
- **output** – The requested image format. The default is "png". For other possible values see *Supported Output Image Formats*.

Raises

ValueError – For unsupported image formats.

Return type

bytes

pdfocr_save(*filename*, *compress=True*, *language='eng'*, *tessdata=None*)

- New in v1.19.0
- Changed in v1.22.5: Support of new parameter for Tesseract's tessdata.

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.
- **tessdata** (*str*) – folder name of Tesseract's language support. If omitted, this information must be present as environment variable `TESSDATA_PREFIX`.

> **Note**
>
> **Will fail** if Tesseract is not installed or if the environment variable "TESSDATA_PREFIX" is not set to the `tessdata` folder name and not provided as parameter.

pdfocr_tobytes(*compress=True*, *language='eng'*, *tessdata=None*)

- New in v1.19.0
- Changed in v1.22.5: Support of new parameter for Tesseract's tessdata.

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=pymupdf.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 = pymupdf.open()
> for imgfile in os.listdir(folder):
>    pix = pymupdf.Pixmap(imgfile)
>    imgpdf = pymupdf.open("pdf", pix.pdfocr_tobytes())
>    doc.insert_pdf(imgpdf)
>    pix = None
>    imgpdf.close()
> doc.save("ocr-images.pdf")
> ```

pil_image()

Create a Pillow Image from the pixmap. PIL / Pillow must be installed.

Raises
ImportError – if Pillow is not installed.

Returns
a `PIL.Image` object

pil_save(**args*, *unmultiply=False*, ***kwargs*)

Write the pixmap as an image file using Pillow. Use this method for output unsupported by MuPDF. Examples are

- Formats 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.webp", optimize=True, dpi=(150, 150))`.

Parameters
unmultiply (*bool*) – If the pixmap's colorspace is RGB with transparency, the alpha values may or may not already be multiplied into the color components ref/green/blue (called "premultiplied"). To enforce undoing premultiplication, set this parameter to `True`. To learn about some background, e.g. look for "Premultiplied alpha" on this page.

For details on other parameters see the Pillow documentation.

Since v1.22.0, PyMuPDF supports JPEG output directly. We recommended to no longer use this method for JPEG output – for performance reasons and for avoiding unnecessary external dependencies.

Raises
ImportError – if Pillow is not installed.

pil_tobytes(**args*, *unmultiply=False*, ***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="WEBP", optimize=True, dpi=(150, 150))`. Also see above. For details on other parameters see the Pillow documentation.

Raises
ImportError – if Pillow is not installed.

Return type
bytes

warp(*quad*, *width*, *height*)

- New in v1.19.3

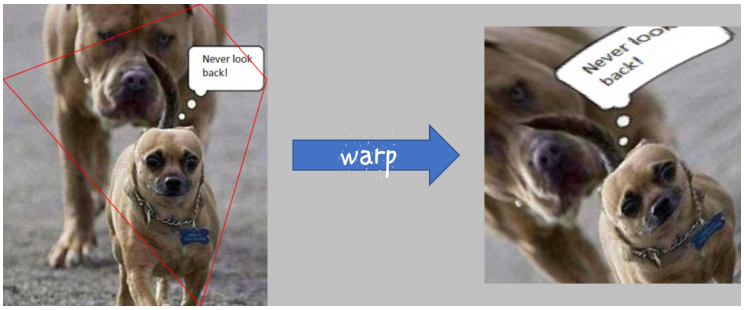
Return a new pixmap by "warping" the quad such that the quad corners become the new pixmap's corners. The target pixmap's *IRect* will be `(0, 0, width, height)`.

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
Pixmap



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

Returns
A tuple `(ratio, pixel)` where `0 < ratio <= 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"x00x00x00")` means that 95% of all pixels are black. See an example here *How to Use Pixmaps: Checking Text Visibility*.

`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)
```

After the Pixmap has been destroyed, any attempt to use the memoryview will fail with ValueError.

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.

Warning: after the Pixmap has been destroyed, the Python pointer will be invalid and attempting to use it may crash the Python interpreter.

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

x

X-coordinate of top-left corner in pixels. Cannot directly be changed – use *Pixmap.set_origin()*.

Type
int

y

Y-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.colorspace.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

27.15.1 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 *pymupdf.open(...)*. The result will then appear as a document containing one single page. Creating a pixmap of this page offers all the options available in this context: apply a matrix, choose colorspace and alpha, confine the pixmap to a clip area, etc.

SVG images are only supported via method 2 above, not directly as pixmaps. But remember: the result of this is a **raster image** as is always the case with pixmaps[1].

27.15.2 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 string identifying the desired format (**Format** column below). Please note that not all combinations of pixmap colorspace, transparency support (alpha) and image format are possible.

[1] If you need a **vector image** from the SVG, you must first convert it to a PDF. Try *Document.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.

Format	Colorspaces	alpha	Extensions	Description
jpg, jpeg	gray, rgb, cmyk	no	.jpg, .jpeg	Joint Photographic Experts Group
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 = pymupdf.Pixmap(pymupdf.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 and JPEG are probably the most popular.
- 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).

This documentation covers all versions up to 1.28.2.

27.16 Point

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

Attribute / Method	Description
`Point.distance_to()`	calculate distance to point or rect
`Point.norm()`	the Euclidean norm
`Point.transform()`	transform point with a matrix
`Point.abs_unit`	same as unit, but positive coordinates
`Point.unit`	point coordinates divided by *abs(point)*
`Point.x`	the X-coordinate
`Point.y`	the Y-coordinate

Class API

class Point

__init__(*self*)

__init__(*self*, *x*, *y*)

__init__(*self*, *point*)

__init__(*self*, *sequence*)

Overloaded constructors.

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

With another point specified, a **new copy** will be created, "sequence" is a Python sequence of 2 numbers (see *Using Python Sequences as Arguments in PyMuPDF*).

Parameters

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

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

Calculate the distance to *x*, which may be *point_like* or *rect_like*. The distance is given in units of either pixels (default), inches, centimeters or millimeters.

Parameters

- **x** (*point_like*,*rect_like*) – to which to compute the distance.
- **unit** (*str*) – the unit to be measured in. One of "px", "in", "cm", "mm".

Return type
float

Returns

the distance to *x*. If this is *rect_like*, then the distance

- is the length of the shortest line connecting to one of the rectangle sides
- is calculated to the **finite version** of it
- is zero if it **contains** the point

norm()

- New in version 1.16.0

Return the Euclidean norm (the length) of the point as a vector. Equals result of function *abs()*.

transform(*m*)

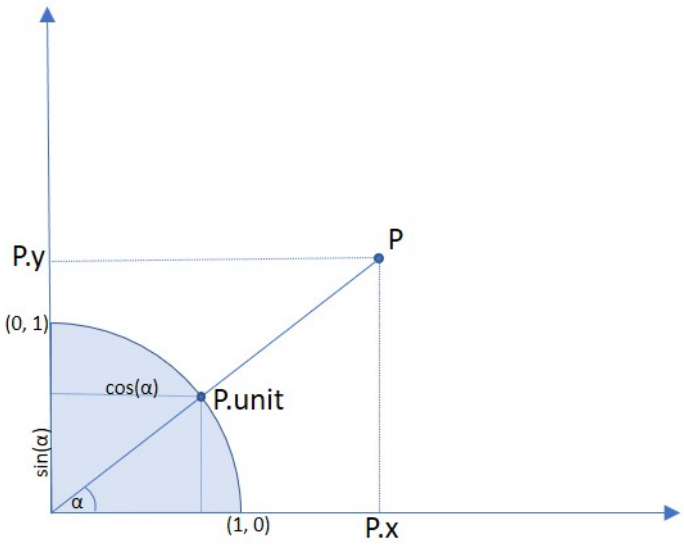
Apply a matrix to the point and replace it with the result.

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

Return type
Point

unit

Result of dividing each coordinate by *norm(point)*, the distance of the point to (0,0). This is a vector of length 1 pointing in the same direction as the point does. Its x, resp. y values are equal to the cosine, resp. sine of the angle this vector (and the point itself) has with the x axis.



Type
Point

abs_unit

Same as `unit` above, replacing the coordinates with their absolute values.

Type
Point

x

The x coordinate

Type
float

y

The y coordinate

Type
float

Note

- This class adheres to the Python sequence protocol, so components can be accessed via their index, too. Also refer to *Using Python Sequences as Arguments in PyMuPDF*.
- Rectangles can be used with arithmetic operators – see chapter *Operator Algebra for Geometry Objects*.

This documentation covers all versions up to 1.28.2.

27.17 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 enveloping rectangle. Vice versa, rectangles now have attributes *Rect.quad*, resp. *IRect.quad* to obtain their respective tetragon versions.

Methods / Attributes	Short Description
Quad.transform()	transform with a matrix
Quad.morph()	transform with a point and matrix
Quad.ul	upper left point
Quad.ur	upper right point
Quad.ll	lower left point
Quad.lr	lower right point
Quad.is_convex	true if quad is a convex set
Quad.is_empty	true if quad is an empty set
Quad.is_rectangular	true if quad is congruent to a rectangle
Quad.rect	smallest containing *Rect*
Quad.width	the longest width value
Quad.height	the longest height value

Class API

class Quad

__init__(*self*)

__init__(*self*, *ul*, *ur*, *ll*, *lr*)

__init__(*self*, *quad*)

__init__(*self*, *sequence*)

Overloaded constructors: "ul", "ur", "ll", "lr" stand for *point_like* objects (the four corners), "sequence" is a Python sequence with four *point_like* objects.

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

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

transform(*matrix*)

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

Parameters

matrix (*matrix_like*) – the matrix.

morph(*fixpoint*, *matrix*)

(New in version 1.17.0) "Morph" the quad with a matrix-like using a point-like as fixed point.

Parameters

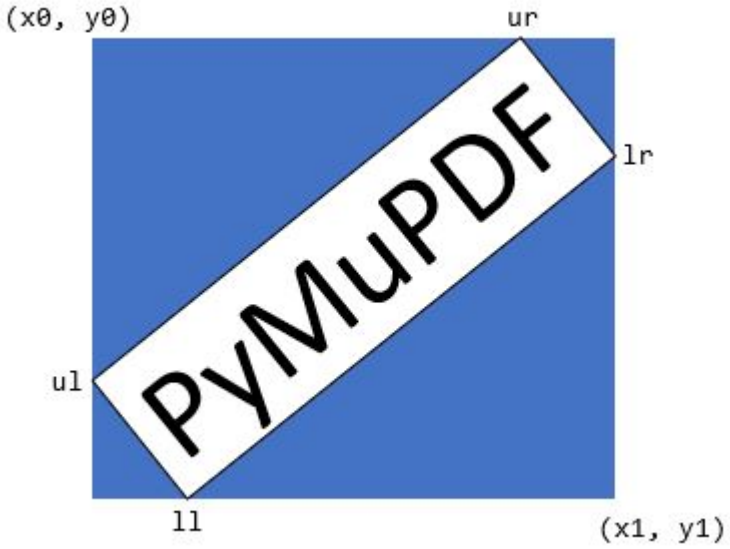
- **fixpoint** (*point_like*) – the point.
- **matrix** (*matrix_like*) – the matrix.

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

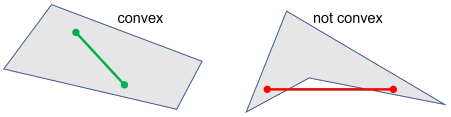
Lower right point.

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

27.17.1 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*.

27.17.2 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
```

This documentation covers all versions up to 1.28.2.

27.18 Rect

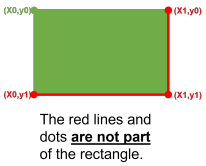
Rect represents a rectangle defined by four floating point numbers x0, y0, x1, y1. They are treated as being coordinates of two diagonally opposite points. The first two numbers are regarded as the "top left" corner $P_{(x0,y0)}$ and $P_{(x1,y1)}$ as the "bottom right" one. However, these two properties need not coincide with their intuitive meanings – read on.

The following remarks are also valid for *IRect* objects:

- 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.
- Units are in points, where 72 points is 1 inch.
- **For any given quadruple of numbers, the geometrically "same" rectangle can be defined in four different ways:**
    1. Rect($P_{(x0,y0)}$, $P_{(x1,y1)}$)
    2. Rect($P_{(x1,y1)}$, $P_{(x0,y0)}$)
    3. Rect($P_{(x0,y1)}$, $P_{(x1,y0)}$)
    4. Rect($P_{(x1,y0)}$, $P_{(x0,y1)}$)

(Changed in v1.19.0) Hence some classification:

- A rectangle is called **valid** if `x0 <= x1` and `y0 <= y1` (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 >= x1` or `y0 >= 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 = FZ_MIN_INF_RECT` and `x1 = y1 = 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 >= x1 or y0 >= y1 – includes invalid rects
valid	n/a	x0 <= x1 and y0 <= y1
infinite	all rects where x0 > x1 or y1 > y0	**exactly one infinite rect / irect!**
coordinate values	all numbers	`FZ_MIN_INF_RECT` <= `number` <= `FZ_MAX_INF_RECT`
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
Rect.contains()	checks containment of point_likes and rect_likes
Rect.get_area()	calculate rectangle area
Rect.include_point()	enlarge rectangle to also contain a point
Rect.include_rect()	enlarge rectangle to also contain another one
Rect.intersect()	common part with another rectangle
Rect.intersects()	checks for non-empty intersections
Rect.morph()	transform with a point and a matrix
Rect.torect()	the matrix that transforms to another rectangle
Rect.norm()	the Euclidean norm
Rect.normalize()	makes a rectangle valid
Rect.round()	create smallest *IRect* containing rectangle
Rect.transform()	transform rectangle with a matrix
Rect.bottom_left	bottom left point, synonym *bl*
Rect.bottom_right	bottom right point, synonym *br*
Rect.height	rectangle height
Rect.irect	equals result of method *round()*
Rect.is_empty	whether rectangle is empty
Rect.is_valid	whether rectangle is valid
Rect.is_infinite	whether rectangle is infinite
Rect.top_left	top left point, synonym *tl*
Rect.top_right	top_right point, synonym *tr*
Rect.quad	*Quad* made from rectangle corners
Rect.width	rectangle width
Rect.x0	left corners' x coordinate
Rect.x1	right corners' x -coordinate
Rect.y0	top corners' y coordinate
Rect.y1	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.

```
>>> pymupdf.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 = pymupdf.Rect(100, 100, 200, 100.001)
>>> r.is_empty  # rect is NOT empty
False
>>> r.round()  # but its irect IS empty!
pymupdf.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_like*) – 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_like*) – 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 `r` is empty, the current rectangle remains unchanged. Else if the current rectangle is empty, it is replaced by `r`.

Parameters
r (*rect_like*) – Second rectangle

`include_point`(*p*)

The smallest rectangle containing the current one and *point_like* `p` is calculated and **replaces the current** one. **The infinite rectangle remains unchanged.** To create the rectangle that wraps a sequence of points, start with *EMPTY_RECT()* and successively include the members of the sequence.

Parameters
p (*point_like*) – 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 *pymupdf.Rect(p1, p2)* and *pymupdf.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

`torect`(*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. See an example use here *How to Use Pixmaps: Checking Text Visibility*.

`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*.

This documentation covers all versions up to 1.28.2.

27.19 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
Shape.commit()	update the page's contents
Shape.draw_bezier()	draw a cubic Bezier curve
Shape.draw_circle()	draw a circle around a point
Shape.draw_curve()	draw a cubic Bezier using one helper point
Shape.draw_line()	draw a line
Shape.draw_oval()	draw an ellipse
Shape.draw_polyline()	connect a sequence of points
Shape.draw_quad()	draw a quadrilateral
Shape.draw_rect()	draw a rectangle
Shape.draw_sector()	draw a circular sector or piece of pie
Shape.draw_squiggle()	draw a squiggly line
Shape.draw_zigzag()	draw a zigzag line
Shape.finish()	finish a set of draw commands
Shape.insert_text()	insert text lines
Shape.insert_textbox()	fit text into a rectangle
Shape.doc	stores the page's document
Shape.draw_cont	draw commands since last *Shape.finish()*
Shape.height	stores the page's height
Shape.lastPoint	stores the current point
Shape.page	stores the owning page
Shape.rect	rectangle surrounding drawings
Shape.text_cont	accumulated text insertions
Shape.totalcont	accumulated string to be stored in *contents*
Shape.width	stores the page's width

Class API

`class Shape`

`__init__`(*self*, *page*)

Create a new drawing. During importing PyMuPDF, the *pymupdf.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 * breadth*. The breadth parameter will

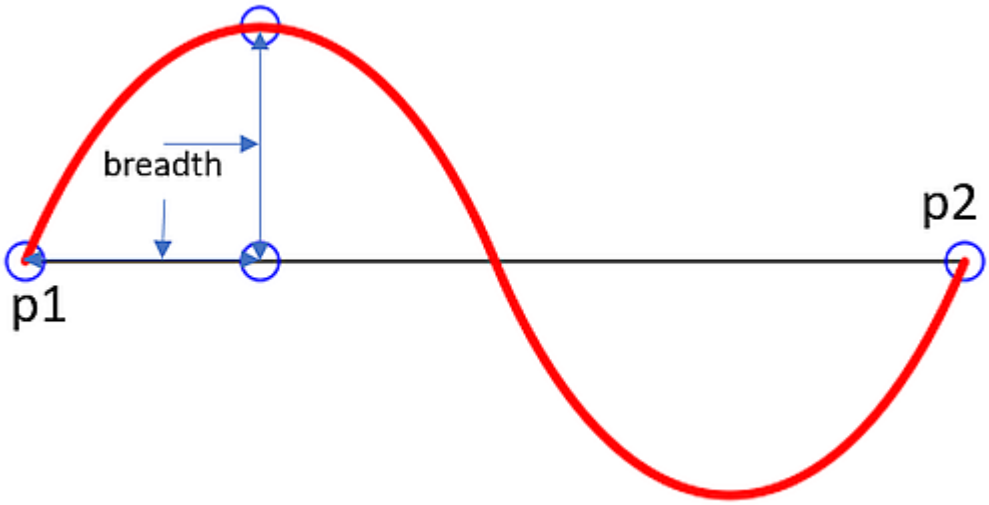
be adjusted as necessary to meet this condition. The drawn line will always turn "left" when leaving *p1* and always join *p2* from the "right".

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.

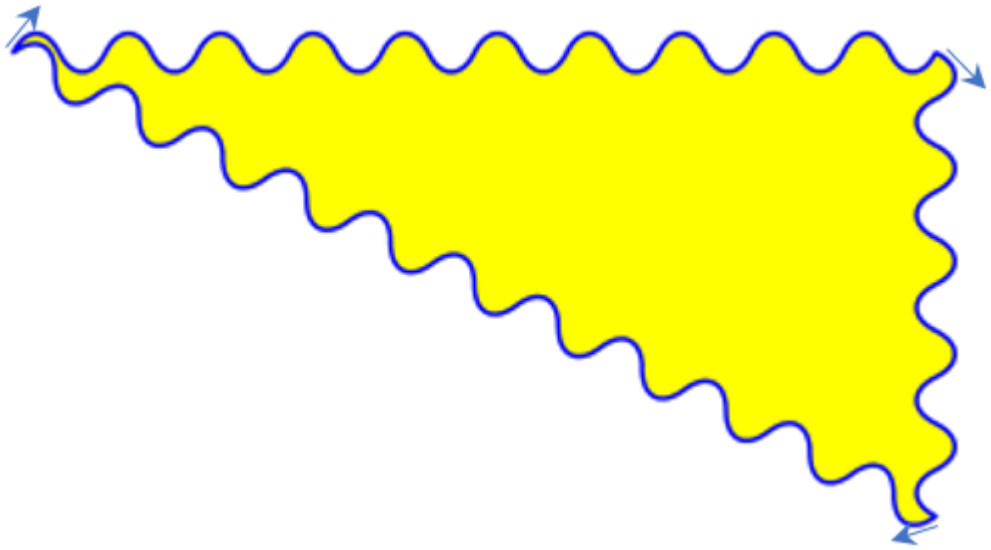
Return type
Point

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 pymupdf
>>> doc=pymupdf.open()
>>> page=doc.new_page()
>>> r = pymupdf.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.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.

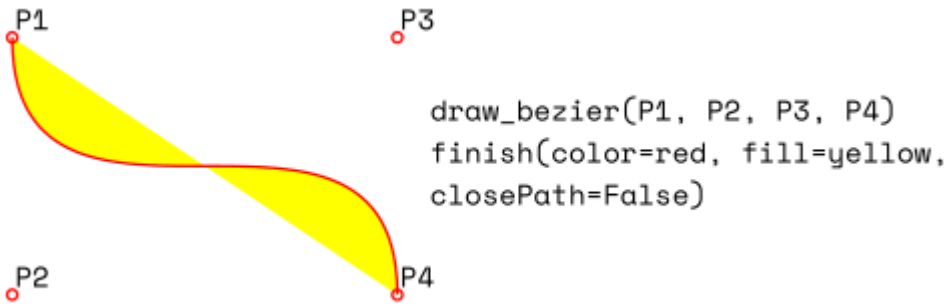
All arguments are *point_like* objects.

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.

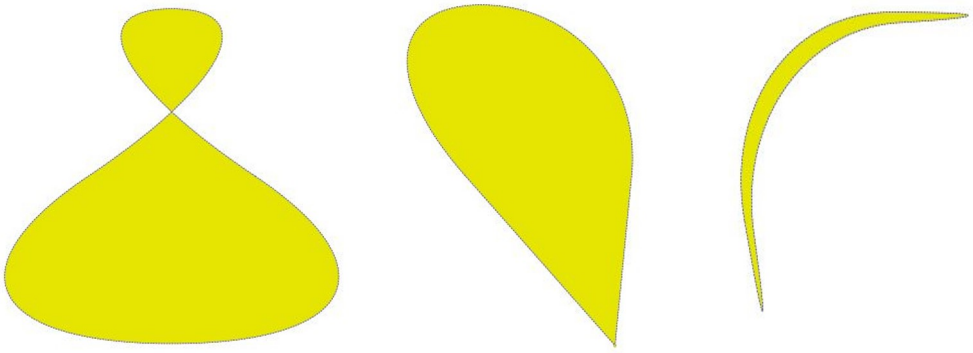
The drawing starts and ends at the middle point of the line `bottom-left -> top-left` corners in an anti-clockwise movement.

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.

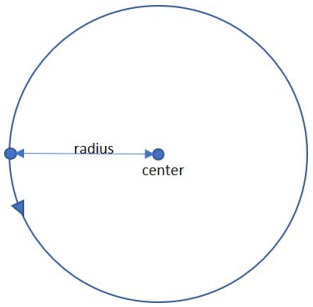
This is a shortcut for `draw_sector(center, start, 360, fullSector=False)`. To draw the same circle in a **clockwise** movement, use `-360` as degrees.

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*)

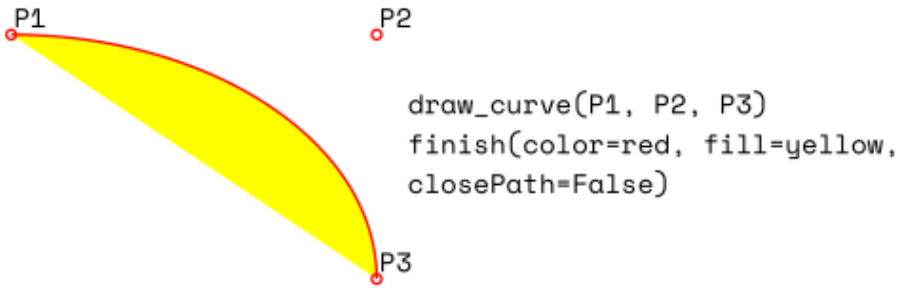
A special case of *draw_bezier()*: Draw a cubic Bezier curve from *p1* to *p3*. On each of the two lines `p1 -> p2` and `p3 -> p2` one control point is generated. Both control points will therefore be on the same side of the line `p1 -> p3`. This guaranties that the curve's curvature does not change 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_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

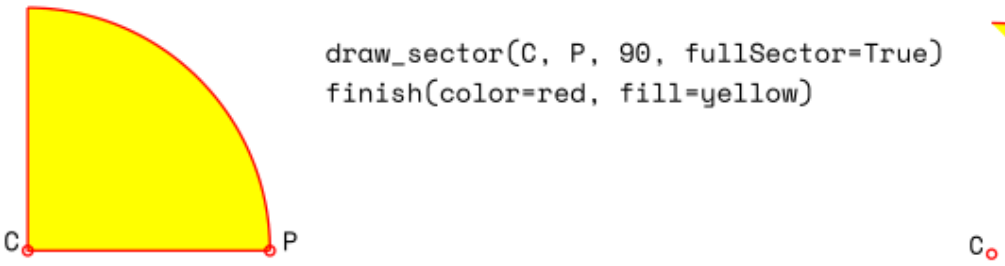
- **center** (*point_like*) – the center of the circle.
- **point** (*point_like*) – one of the two end points of the pie's arc segment. The other one is calculated from the *angle*.
- **angle** (*float*) – the angle of the sector in degrees. Used to calculate the other end point of the arc. Depending 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_rect(*rect*, *, *radius=None*)

- Changed in v1.22.0: Added parameter *radius*.

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.
- **radius** (*multiple*) – draw rounded rectangle corners. If not `None`, specifies the radius of the curvature as a percentage of a rectangle side length. This must one or (a tuple of) two floats `0 < radius <= 0.5`, where 0.5 corresponds to 50% of the respective side. If a float, the radius of the curvature is computed as `radius * min(width, height)`, drawing the corner's perimeter as a quarter circle. If a tuple

(rx, ry) is given, then the curvature is asymmetric with respect to the horizontal and vertical directions. A value of radius=(0.5, 0.5) draws an ellipse.

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=(0,)*, *fill=None*, *lineCap=0*, *lineJoin=0*, *dashes=None*, *closePath=True*, *even_odd=False*, *morph=(fixpoint, matrix)*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

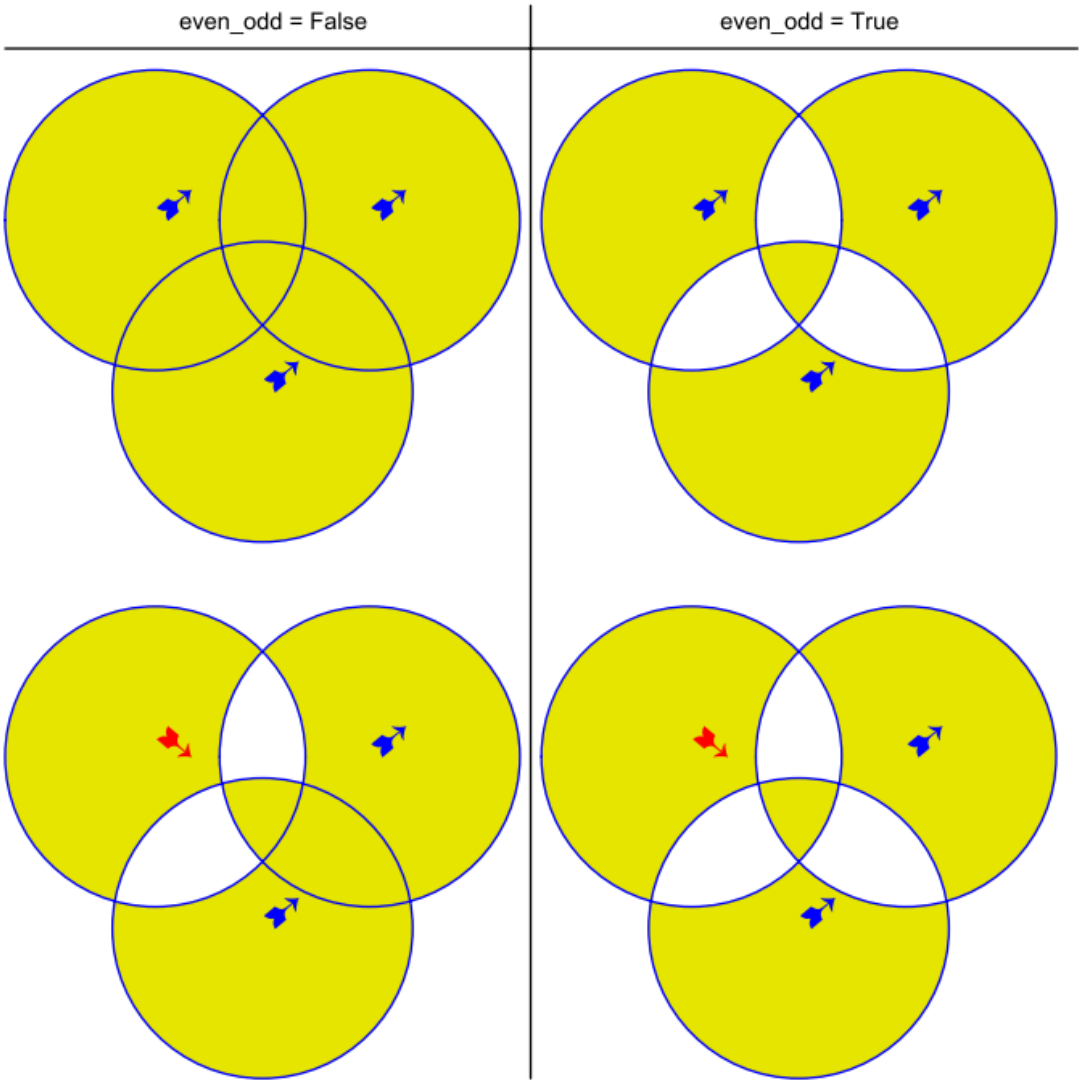
Finish a set of *draw*()* methods by applying *Common Parameters* to all of them.

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.

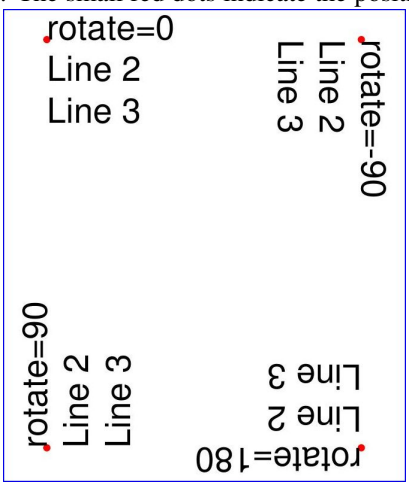
Of the four shapes in above image, the top two each show three circles drawn in standard manner (anti-clockwise, look at the arrows). The lower two shapes contain one (the top-left) circle drawn clockwise. As can be seen, area orientation is irrelevant for the right column (even-odd rule).

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*, *miter_limit=1*, *border_width=1*, *rotate=0*, *morph=None*, *stroke_opacity=1*, *fill_opacity=1*, *oc=0*)

Insert text lines starting 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 (the **border line** of a character). Only `0 <= value <= 1` will be considered. 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*, ***, *align=TEXT_ALIGN_LEFT*, *border_width=1*, *color=None*, *encoding=TEXT_ENCODING_LATIN*, *expandtabs=8*, *fill=None*, *fill_opacity=1*, *fontfile=None*, *fontname='helv'*, *fontsize=11*, *lineheight=None*, *miter_limit=1*, *morph=None*, *oc=0*, *render_mode=0*, *rotate=0*, *set_simple=False*, *stroke_opacity=1*)

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

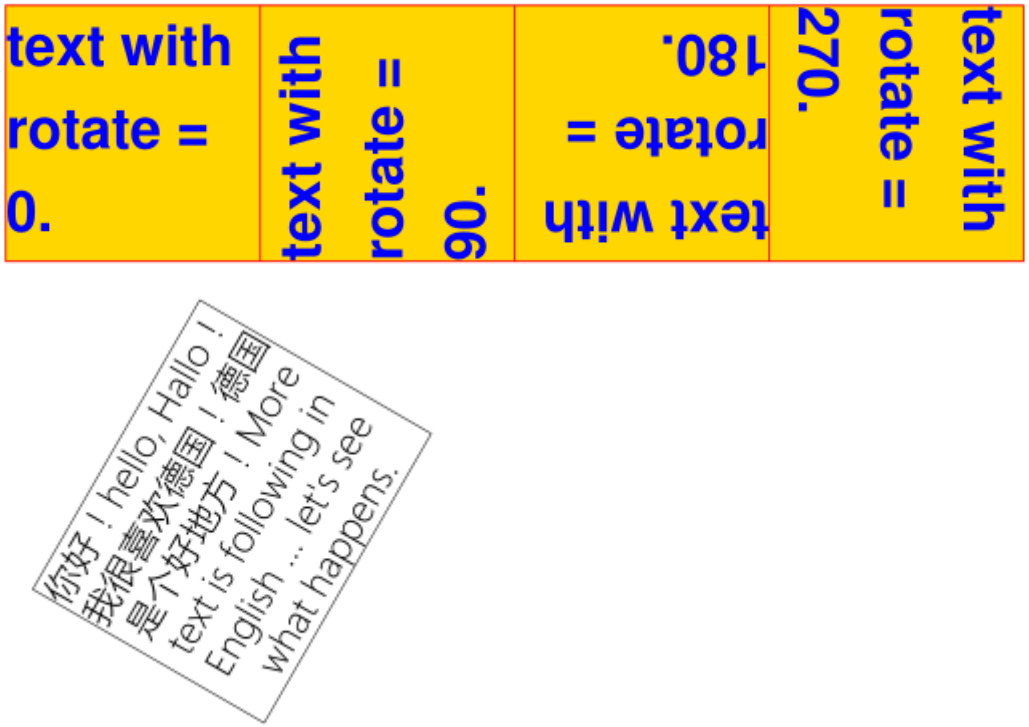
- **rect** (*rect_like*) – the area to use. It must be finite and not empty.
- **buffer** (*str/sequence*) – the text to be inserted. Must be specified as a string or a sequence of strings. Line breaks are respected also when occurring in a sequence entry.
- **align** (*int*) – align each text line. 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**.
- **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.
- **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.
- **oc** (*int*) – *(new in v1.18.4)* the *xref* number of an *OCG* or *OCMD* to make this text conditionally displayable.
- **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, the four values `0`, `90`, `180` and `270` (= `-90`) are processed, 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.
- **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).

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 - pymupdf.Rect(point, point)) * ~matrix + pymupdf.
↪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

27.19.1 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.

27.19.2 Examples

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

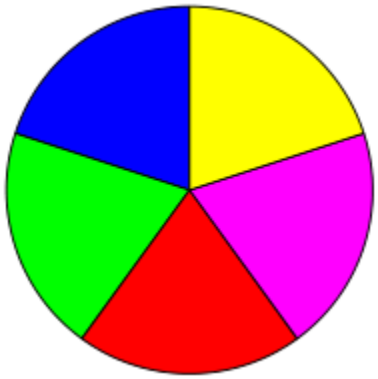
```
shape = page.new_shape()  # start a new shape
cols = (...)  # a sequence of RGB color triples
pieces = len(cols)  # number of pieces to draw
beta = 360. / pieces  # angle of each piece of pie
center = pymupdf.Point(...)  # center of the pie
p0 = pymupdf.Point(...)  # starting point
```

(continues on next page)

(continued from previous page)

```
for i in range(pieces):
    p0 = shape.draw_sector(center, p0, beta,
                           fullSector=True) # draw piece
    # now fill it but do not connect ends of the arc
    shape.finish(fill=cols[i], closePath=False)
shape.commit()  # update the page
```

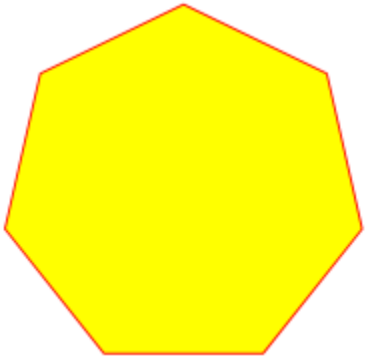
Here is an example for 5 colors:



2. Create a regular n-edged polygon (fill yellow, red border). We use *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 = pymupdf.Point(...)  # center of circle
p0 = pymupdf.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:



27.19.3 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, see: `fontsize`.

---

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 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 *pymupdf.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.

---

width (*float*)

The stroke ("border") width of the elements in a shape (if applicable). The default value is 1. The values width, color and fill have the following relationship / dependency:

- If `fill=None` shape elements will always be drawn with a border - even if `color=None` (in which case black is taken) or `width=0` (in which case 1 is taken).
- Shapes without border can only be achieved if a fill color is specified (which may be white of course). To achieve this, specify `width=0`. In this case, the `color` parameter is ignored.

---

stroke_opacity / fill_opacity (*floats*)

Both values are floats in range [0, 1]. Negative values or values > 1 will ignored (in most cases). Both set the transparency such that a value 0.5 corresponds to 50% transparency, 0 means invisible and 1 means intransparent. For e.g. a rectangle the stroke opacity applies to its border and fill opacity to its interior.

For text insertions (`Shape.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.

---

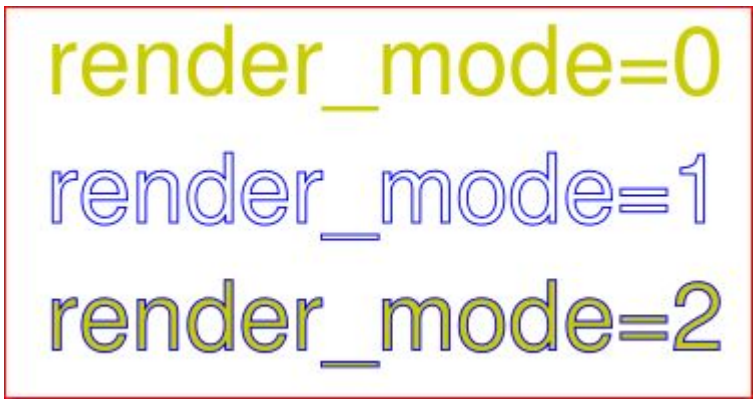
render_mode (*int*)

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.



---

miter_limit (*float*)

A float specifying the maximum acceptable value of the quotient `miter-length / line-width` ("miter quotient"). Used in text output methods. This is only relevant for non-zero render mode values – then, characters are written with border lines (i.e. "stroked").

If two lines stroking some character meet at a sharp (<= 90°) angle and the line width is large enough, then "spikes" may become visible – causing an ugly appearance as shown below. For more background, see page 126 of the *Adobe PDF References*.

For instance, when joins meet at 90°, then the miter length is `sqrt(2) * line-width`, so the miter quotient is `sqrt(2)`.

If `miter_limit` is exceeded, then all joins with a larger qotient will appear as beveled ("butt" appearance).

The default value 1 (and any smaller value) will ensure that all joins are rendered as a butt. A value of `None` will use the PDF default value.

Example text showing spikes (`miter_limit=None`):

Example text suppressing spikes (`miter_limit=1`):

---

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 programmer'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).

This documentation covers all versions up to 1.28.2.

27.20 Story

- New in v1.21.0

Method / Attribute	Short Description
Story.reset()	"rewind" story output to its beginning
Story.place()	compute story content to fit in provided rectangle
Story.draw()	write the computed content to current page
Story.element_positions()	callback function logging currently processed story content
Story.body	the story's underlying *body*
Story.write()	places and draws Story to a DocumentWriter
Story.write_stabilized()	iterative layout of html content to a DocumentWriter
Story.write_with_links()	like *write()* but also creates PDF links
Story.write_stabilized_with_links()	like *write_stabilized()* but also creates PDF links
Story.fit()	Finds optimal rect that contains the story `self`.
Story.fit_scale()	
Story.fit_height()	
Story.fit_width()	

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 handled for you using the *write()* or `write_stabilised()` 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). If provided, not a complete HTML document is needed. The in-built source parser will forgive (many / most) HTML syntax errors and also accepts HTML fragments like `"<b>Hello, <i>World!</i></b>"`.
- **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 = pymupdf.Story(archive=pymupdf.Archive("myfolder"))`, one can also shorter write `story = pymupdf.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()*.

Story will pass position information to *function*. This information can for example be used to generate a Table of Contents.

Parameters

- **function** (*callable*) – a Python function accepting an `ElementPosition` object. It will be invoked by the Story object to process positioning information. The function **must** be a callable accepting exactly one argument.
- **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. 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.

class FitResult

The result from a `Story.fit*()` method.

Members:

big_enough:
`True` if the fit succeeded.

filled:
From the last call to *Story.place()*.

more:
`False` if the fit succeeded.

numcalls:
Number of calls made to `self.place()`.

parameter:
The successful parameter value, or the largest failing value.

Rect:
The rect created from `parameter`.

fit(*self*, *fn*, *pmin=None*, *pmax=None*, *delta=0.001*, *verbose=False*)

Finds optimal rect that contains the story `self`.

Returns a *Story.FitResult* instance.

On success, the last call to `self.place()` will have been with the returned rectangle, so `self.draw()` can be used directly.

Parameters

- **fn** – A callable taking a floating point `parameter` and returning a `pymupdf.Rect()`. If the rect is empty, we assume the story will not fit and do not call `self.place()`.

  Must guarantee that `self.place()` behaves monotonically when given rect `fn(parameter)` as `parameter` increases. This usually means that both width and height increase or stay unchanged as `parameter` increases.
- **pmin** – Minimum parameter to consider; `None` for -infinity.
- **pmax** – Maximum parameter to consider; `None` for +infinity.
- **delta** – Maximum error in returned `parameter`.
- **verbose** – If true we output diagnostics.

fit_scale(*self*, *rect*, *scale_min=0*, *scale_max=None*, *delta=0.001*, *verbose=False*)

Finds smallest value `scale` in range `scale_min..scale_max` where `scale * rect` is large enough to contain the story `self`.

Returns a *Story.FitResult* instance.

Parameters

- **width** – width of rect.
- **height** – height of rect.
- **scale_min** – Minimum scale to consider; must be >= 0.
- **scale_max** – Maximum scale to consider, must be >= scale_min or `None` for infinite.
- **delta** – Maximum error in returned scale.
- **verbose** – If true we output diagnostics.

fit_height(*self*, *width*, *height_min=0*, *height_max=None*, *origin=(0, 0)*, *delta=0.001*, *verbose=False*)

Finds smallest height in range `height_min..height_max` where a rect with size `(width, height)` is large enough to contain the story `self`.

Returns a *Story.FitResult* instance.

Parameters

- **width** – width of rect.
- **height_min** – Minimum height to consider; must be >= 0.
- **height_max** – Maximum height to consider, must be >= height_min or `None` for infinite.
- **origin** – `(x0, y0)` of rect.
- **delta** – Maximum error in returned height.

- **verbose** – If true we output diagnostics.

fit_width(*self*, *height*, *width_min=0*, *width_max=None*, *origin=(0, 0)*, *delta=0.001*, *verbose=False*)

Finds smallest width in range `width_min..width_max` where a rect with size `(width, height)` is large enough to contain the story `self`.

Returns a *Story.FitResult* instance.

Parameters

- **height** – height of rect.
- **width_min** – Minimum width to consider; must be >= 0.
- **width_max** – Maximum width to consider, must be >= width_min or `None` for infinite.
- **origin** – `(x0, y0)` of rect.
- **delta** – Maximum error in returned width.
- **verbose** – If true we output diagnostics.

27.20.1 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 = pymupdf.paper_rect("letter")  # size of a page
WHERE = MEDIABOX + (36, 36, -36, -36)  # leave borders of 0.5 inches
story =  pymupdf.Story(html=HTML)  # make the story
writer = pymupdf.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
```

Attributes of the ElementPosition class

Exactly one parameter must be passed to the function provided by *Story.element_positions()*. It is an object with the following attributes:

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 `pymupdf.Story.write*()` functions.

This documentation covers all versions up to 1.28.2.

27.21 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
extractText()	extract plain text	"text"
extractTEXT()	synonym of previous	"text"
extractBLOCKS()	plain text grouped in blocks	"blocks"
extractWORDS()	all words with their bbox	"words"
extractHTML()	page content in HTML format	"html"
extractXHTML()	page content in XHTML format	"xhtml"
extractXML()	page text in XML format	"xml"
extractDICT()	page content in *dict* format	"dict"
extractJSON()	page content in JSON format	"json"
extractRAWDICT()	page content in *dict* format	"rawdict"
extractRAWJSON()	page content in JSON format	"rawjson"
search()	Search for a string in the page	*Page.search_for()*

Class API

class TextPage

extractText(*sort=False*)

extractTEXT(*sort=False*)

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.

Parameters
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.

Return type
str

extractBLOCKS()

Textpage content as a list of text lines grouped by block. Each list items looks like this:

```
``(x0, y0, x1, y1, "lines in the block", block_no, block_type)``
```

The first four entries are the block's bbox coordinates, *block_type* is 1 for an image block, 3 for a vector block, and 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**. Image blocks are included only if the extraction flag bit *TEXT_PRESERVE_IMAGES* is set. An image block tuple will look like this:

```
``(x0, y0, x1, y1, "<image: colorspace-name, w: width, h: height, bpc: bits_
↪per_component>\n", block_no, 1)``
```

For a **vector block**, the following item will be included. Vector blocks are included only if the extraction flag bit *TEXT_COLLECT_VECTORS* is set. A vector block tuple will look like this:

```
``(x0, y0, x1, y1, "<vector stroked, color: #rrggbb, alpha: 255, is-rect: true,
↪ continues: false>\n", block_no, 3)``
```

The keyword "vector" is followed by either "stroked" or "filled". The color is given in HTML (hexadecimal RGB) format. Property `is-rect` is true, if the vector is not a curve and parallel to the x- or y-axis. So in essence is either a real rectangle or a line segment. Property `continues` indicates whether the vector is part of a path (and not the first item).

Note

When no further details are needed (as provided by *Page.get_drawings()*), then this is an **inexpensive** way to extract basic vector graphics information. Another major advantage is that all block types (text, images and vectors) are included in the output in the same order as they are present in the page's *contents* stream.

This is a high-speed method with just enough information to output plain text in desired reading sequence.

Return type
list

extractWORDS(*delimiters=None*)

- Changed in v1.23.5: added `delimiters` parameter

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)
```

Parameters

delimiters (*str*) – (new in v1.23.5) use these characters as *additional* word separators. By default, all white spaces (including the non-breaking space `0xA0`) indicate start and end of a word. Now you can specify more characters causing this. For instance, the default will return `"john.doe@outlook.com"` as **one** word. If you specify `delimiters="@."` then the **four** words `"john"`, `"doe"`, `"outlook"`, `"com"` will be returned. Other possible uses include ignoring punctuation characters `delimiters=string.punctuation`. The "word" strings will not contain any delimiting character.

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(*sort=False*)

Textpage content as a Python dictionary. Provides same information detail as HTML. See below for the structure.

Parameters

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.

Return type

dict

extractJSON(*sort=False*)

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.

Parameters

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.

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(*sort=False*)

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.

Parameters
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.

Return type
dict

extractRAWJSON(*sort=False*)

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.

Parameters
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.

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".

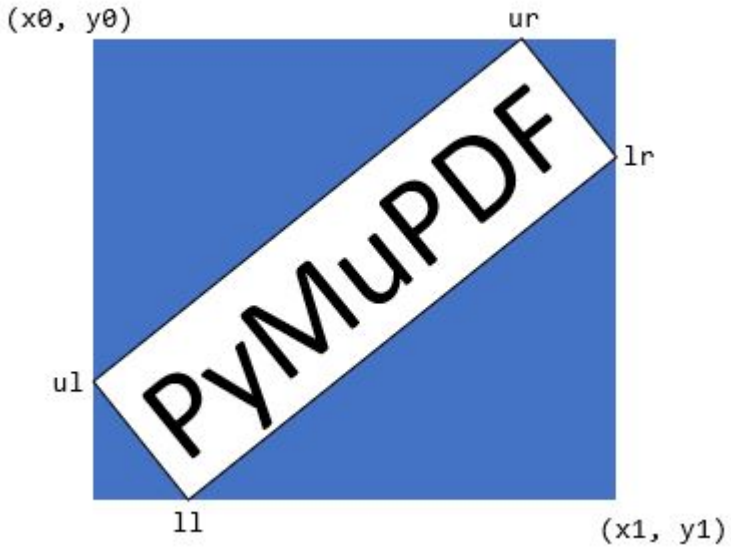
Note

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.

27.21.1 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.

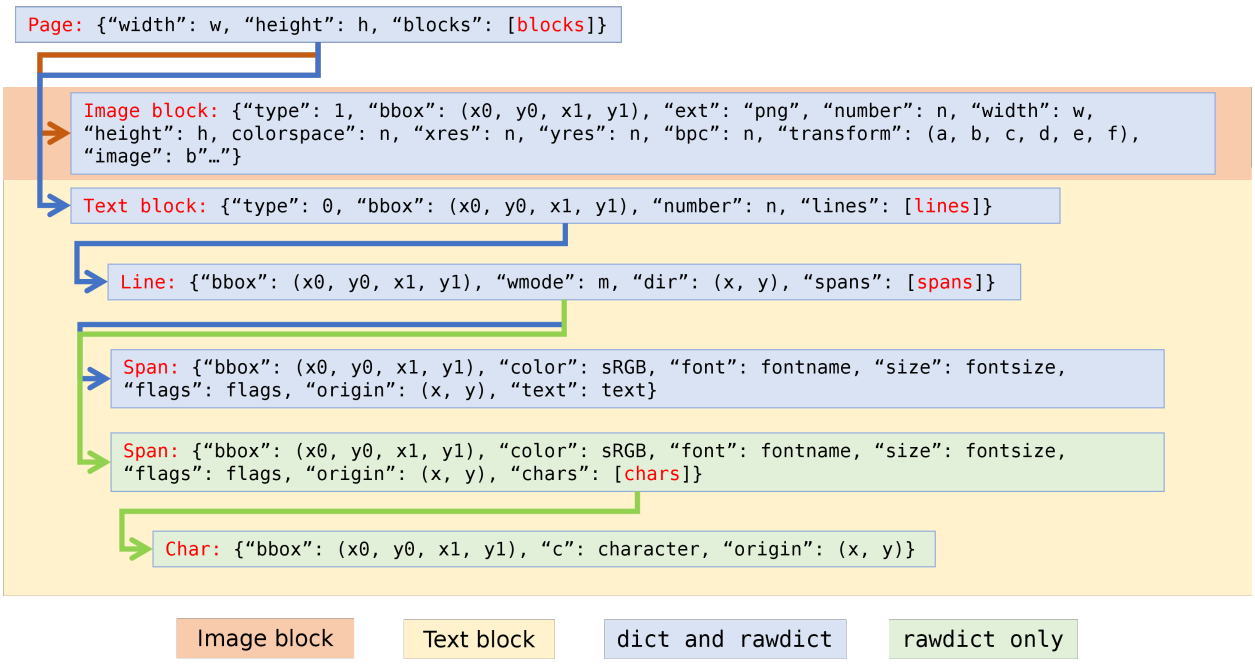
In the vast majority of cases, we are dealing with **horizontal text only**, where bboxes provide entirely sufficient information.

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



Page Dictionary

Key	Value
width	width of the `clip` rectangle *(float)*
height	height of the `clip` rectangle *(float)*
blocks	*list* of block dictionaries

Block Dictionaries

Block dictionaries come in different formats for **vector blocks**, **image blocks** and **text blocks**. Vector blocks are included only if the extraction flag bit *TEXT_COLLECT_VECTORS* is set. Image blocks are included only if the extraction flag bit *TEXT_PRESERVE_IMAGES* is set.

Vector block:

Key	Value
type	3 = vector (`int`)
bbox	vector bbox on page (*rect_like*)
number	block count (`int`)
stroked	either stroked (`True`) or filled (`False`) (`bool`)
isrect	whether the vector is axis-parallel (`bool`). Can be a line or a rectangle. Curves or diagonal lines are `False`.
continues	whether the vector is (not the last) part of a sequence of vectors in a *path* (`bool`).
color	sRGB integer, e.g. 0xRRGGBB (`int`).
alpha	Transparency, a value in `range(256)` (`int`).

This information is a true subset of the output of *Page.get_drawings()*. Its advantage is its speed (because it is extracted alongside one *TextPage* creation) and the fact that vector blocks are included in the overall page content sequence together with text and images.

Image block:

Key	Value
type	1 = image (`int`)
bbox	image bbox on page (*rect_like*)
number	block count (`int`)
ext	image type (`str`), as file extension, see below
width	original image width (`int`)
height	original image height (`int`)
colorspace	colorspace component count (`int`)
xres	resolution in x-direction (`int`)[3]
yres	resolution in y-direction (`int`)[3]
bpc	bits per component (`int`)
transform	matrix transforming image rect to bbox (*matrix_like*)
size	size of the image in bytes (`int`)
image	image content (`bytes`)
mask	image mask content (`bytes`) for transparent images

[3] This value is always 96, the default of the PDF interpreter. It **does not reflect** the resolution of the image itself. If you need the image's resolution, use the *Pixmap.xres()* and *Pixmap.yres()* attributes of the *Pixmap* created from the returned image binary.

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 == pymupdf.Rect(0, 0, 1, 1)` is true, lookup details here: *Image Transformation Matrix*.
4. A transparent image may be accompanied by a mask image. This is stored under key `"mask"` and has the format of a `DeviceGray` PNG image. Otherwise the value of this key is `None`. If present, you may be able to recover (an equivalent of) the original image – i.e. with transparency – by creating *Pixmap* objects from the "image", respectively "mask" values and overlay them. This is not guaranteed to always work because mask images come in multiple formats, of which not all qualify for the conditions under which overlaying Pixmaps are supported. Here is a code snippet:

```
>>> base = pymupdf.Pixmap(block["image"])
>>> mask = pymupdf.Pixmap(block["mask"])
>>> result = pymupdf.Pixmap(base, mask)
```

Text block:

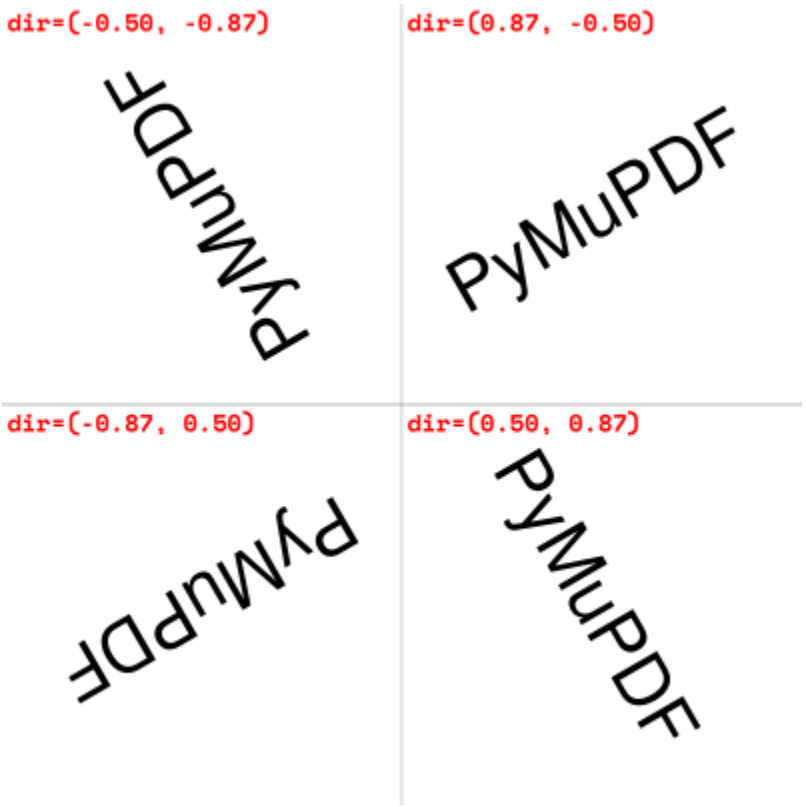
Key	Value
type	0 = text *(int)*
bbox	block rectangle, *rect_like*
number	block count *(int)*
lines	*list* of text line dictionaries

[1] Image specifications for a PDF page are done in a page's (sub-) *dictionary*, called `/Resources`. Resource dictionaries can be **inherited** from any of the page's parent objects (usually the *catalog* – the top-level parent). The PDF creator may e.g. define one `/Resources` on file level, naming all images and / or all fonts ever used by any page. In these cases, *Page.get_images()* and *Page.get_fonts()* will consequently return the same lists for all pages. If desired, this situation can be reverted using *Page.clean_contents()*. After execution, the page's object definition will show fonts and images that are actually used.

Line Dictionary

Key	Value
bbox	line rectangle, *rect_like*
wmode	writing mode *(int)*: 0 = horizontal, 1 = vertical
dir	writing direction, *point_like*
spans	*list* of span dictionaries

The value of key *"dir"* is the **unit vector** `dir = (cosine, -sine)` of the angle, which the text has relative to the x-axis[2]. 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.



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.

[2] The coordinate systems of MuPDF and PDF are different in that MuPDF uses the page's top-left point as `(0, 0)`. In PDF, this is the bottom-left point. Therefore, the positive direction for MuPDF's y-axis is **from top to bottom**. This causes the sign change for the sine value here: a **negative** value indicates anti-clockwise rotation of the text.

Key	Value
bbox	span rectangle, *rect_like*
origin	the first character's origin, *point_like*
font	font name *(str)*
ascender	ascender of the font *(float)*
descender	descender of the font *(float)*
size	font size *(float)*
flags	font characteristics *(int)*
char_flags	char characteristics *(int)*
color	text color in sRGB format 0xRRGGBB *(int)*.
alpha	text opacity 0..255 *(int)*.
text	(only for `extractDICT()`) text *(str)*
chars	(only for `extractRAWDICT()`) *list* of character dictionaries

(New in version 1.25.3.0): Added *"alpha"* item.

(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 = pymupdf.Rect(span["bbox"])
>>> o = pymupdf.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 OCRed 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 `pymupdf.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 (*TEXT_FONT_SUPERSCRIPT*) – not a font property, detected by MuPDF code.
- bit 1: italic (*TEXT_FONT_ITALIC*)
- bit 2: serifed (*TEXT_FONT_SERIFED*)
- bit 3: monospaced (*TEXT_FONT_MONOSPACED*)
- bit 4: bold (*TEXT_FONT_BOLD*)

Test these characteristics like so:

```
>>> if flags & pymupdf.TEXT_FONT_BOLD & pymupdf.TEXT_FONT_ITALIC:
        print(f"{span['text']=} is bold and italic")
```

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.

"char_flags" is an integer, which represents extra character properties:

- bit 0: strikeout.
- bit 1: underline.
- bit 2: synthetic (always 0, see char dictionary).
- bit 3: filled.
- bit 4: stroked.
- bit 5: clipped.

For example if not filled and not stroked (`if not (char_flags & 2**3 & 2**4): ...`) then the text will be invisible.

(`char_flags` is new in v1.25.2.)

Character Dictionary for `extractRAWDICT()`

Key	Value
origin	character's left baseline point, *point_like*
bbox	character rectangle, *rect_like*
synthetic	bool.
c	the character (unicode)

(`synthetic` is new in v1.25.3.)

This image shows the relationship between a character's bbox and its quad:

This documentation covers all versions up to 1.28.2.

27.22 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
append()	Add text in horizontal write mode
appendv()	Add text in vertical write mode
fill_textbox()	Fill rectangle (horizontal write mode)
write_text()	Output TextWriter to a PDF page
color	Text color (can be changed)
last_point	Last written character ends here
opacity	Text opacity (can be changed)
rect	Page rectangle used by this TextWriter
text_rect	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, `pymupdf.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 = pymupdf.open()
>>> page = doc.new_page()
>>> text = "PyMuPDF: the Python bindings for MuPDF"
>>> font = pymupdf.Font("figo")  # choose a font with small caps
>>> tw = pymupdf.TextWriter(page.rect)
>>> tw.append((50,100), text, font=font, small_caps=True)
>>> tw.write_text(page)
>>> doc.ez_save("x.pdf")
```

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, `pymupdf.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 `pymupdf.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 (`None`), warn (`True`), or raise an exception (`False`). Overflow text will never be written. **Changed in v1.18.9:**
  - Default is `None`.
  - The list of overflow lines will be returned.
  - When `None` or `True` the partial text will be written.
  - When `False`, and an exception is raised, the text *will not* be written.
- **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).

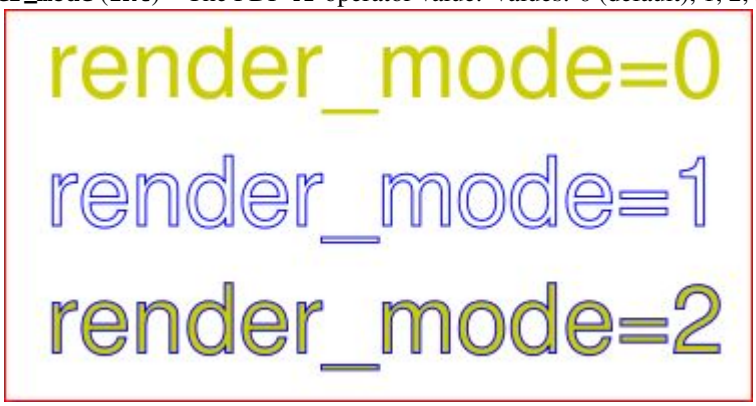
> **Note**
>
> Use these methods as often as is required – there is no technical limit (except memory constraints of your system). You can also mix *append()* and text boxes and have multiple of both. Text positioning is exclusively controlled by the insertion point. Therefore there is no need to adhere 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 (modifiable).

Return type
float

`color`

The text color (modifiable).

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 the TextWriter demo scripts.

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.

This documentation covers all versions up to 1.28.2.

27.23 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
Tools.gen_id()	generate a unique identifier
Tools.store_shrink()	shrink the storables cache[1]
Tools.mupdf_warnings()	return the accumulated MuPDF warnings
Tools.mupdf_display_errors()	control whether MuPDF errors are displayed as messages.
Tools.mupdf_display_warnings()	control whether MuPDF warnings are displayed as messages.
Tools.reset_mupdf_warnings()	empty MuPDF warnings/errors message buffer.
Tools.set_aa_level()	set the anti-aliasing values
Tools.set_annot_stem()	set the prefix of new annotation / link ids
Tools.set_small_glyph_heights()	search and extract using small bbox heights
Tools.set_subset_fontnames()	control suppression of subset fontname tags
Tools.show_aa_level()	return the anti-aliasing values
Tools.unset_quad_corrections()	disable PyMuPDF-specific code
Tools.fitz_config	configuration settings of PyMuPDF
Tools.store_maxsize	maximum storables cache size
Tools.store_size	current storables cache size

Class API

`class Tools`

`gen_id()`

A convenience method returning a unique positive integer which will increase by 1 on every invocation. Example usages include creating unique keys in databases - its creation should be faster than using timestamps by an order of magnitude.

> **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.

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.

[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 pymupdf*), 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.

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*)

Control whether MuPDF errors should be displayed as messages.

Parameters
value

- If None, the current setting is left unchanged.
- Otherwise changes the current setting to bool(value); if True, future MuPDF errors will be shown as *messages*.
- Regardless of this setting, MuPDF errors will always be stored in the warnings store.
- Upon import of this value is True.

Returns
The current setting as True or False.

- New in version 1.16.8

mupdf_display_warnings(*value=None*)

Control whether MuPDF warnings should be displayed as messages.

Parameters
value

- If None, the current setting is left unchanged.
- Otherwise changes the current setting to bool(value); if True, future MuPDF warnings will be shown as *messages*.
- Regardless of this setting, MuPDF warnings will always be stored in the warnings store.
- Upon import of this value is True.

Returns
The current setting as True or False.

- New in version 1.16.8

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 colorspcae rendering
plotter-n	overprint rendering
pdf	PDF documents
xps	XPS documents
svg	SVG documents
cbz	CBZ documents
img	IMG documents
html	HTML documents
epub	EPUB documents
jpx	JPEG2000 images
js	JavaScript
tofu	all TOFU fonts
tofu-cjk	CJK font subset (China, Japan, Korea)
tofu-cjk-ext	CJK font extensions
tofu-cjk-lang	CJK font language extensions
tofu-emoji	TOFU emoji fonts
tofu-historic	TOFU historic fonts
tofu-symbol	TOFU symbol fonts
tofu-sil	TOFU SIL fonts
icc	ICC profiles
py-memory	using Python memory management[2]
base14	Base-14 fonts (should always be true)

For an explanation of the term "TOFU" see this Wikipedia article:

```
In [1]: import pymupdf
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,
```

(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-cjk-ext': False,
'tofu-cjk-lang': False,
'tofu-emoji': False,
'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. 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 function. It will (automatically) decrease only when *Tools.store_maxsize* is going to be exceeded: in this case, will evict low-usage objects until the value is again in range.

Return type
int

27.23.1 Example Session

```
>>> import pymupdf
# print the maximum and current cache sizes
>>> pymupdf.TOOLS.store_maxsize
268435456
>>> pymupdf.TOOLS.store_size
0
>>> doc = pymupdf.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)
>>> pymupdf.TOOLS.store_size
454519
# release (at least) 50% of the storage
>>> pymupdf.TOOLS.store_shrink(50)
13471
>>> pymupdf.TOOLS.store_size
13471
# get a few unique numbers
>>> pymupdf.TOOLS.gen_id()
1
>>> pymupdf.TOOLS.gen_id()
```

(continues on next page)

(continued from previous page)

```
2
>>> pymupdf.TOOLS.gen_id()
3
# close document and see how much cache is still in use
>>> doc.close()
>>> pymupdf.TOOLS.store_size
0
>>>
```

This documentation covers all versions up to 1.28.2.

27.24 Widget

This class represents a PDF Form field "widget". Widgets 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.

Like annotations, widgets also lose connection to their page when the page becomes unavailable, please see here for details. This is relevant especially when updating the widget: this will fail if the original page object is no longer available.

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. The following example shows that the "selected" value is "Male":

```
>>> print(field.field_name, field.button_states())
Gender Second person {'down': ['Male', 'Off'], 'normal': [
↪'Male', 'Off']}
```

on_state()

- New in version 1.22.2

Return the value of the "ON" state of check boxes and radio buttons. For check boxes this is always the value "Yes". For radio buttons, this is the value to select / activate the button.

returns

the value that sets the button to "selected". For non-checkbox, non-radiobutton fields, always `None` is returned. For check boxes the return is `True`. For radio buttons this is the value "Male" in the following example:

```
>>> print(field.field_name, field.button_states())
Gender Second person {'down': ['Male', 'Off'], 'normal': [
↪'Male', 'Off']}
>>> print(field.on_state())
Male
```

So for check boxes and radio buttons, the recommended method to set them to "selected", or to check the state is the following:

```
>>> field.field_value = field.on_state()
>>> field.field_value == field.on_state()
True
```

update(*sync_flags=False*)

After any changes to a widget, this **method must be used** to reflect changes in the PDF[1].

Parameters

sync_flags (*bool*) – if `True`, the widget's *Widget.field_flags* are copied to the `Parent` object (if present) and all widgets named in its `Kids` array. This provides a convenient way to – for example – set all instances of the widget to read-only, no matter on which page they may occur[2].

reset()

Reset the field's value to its default – if defined – or remove it. Do not forget to issue *update()* afterwards.

next

Point to the next form field on the page. The last widget returns `None`.

border_color

A list of up to 4 floats defining the field's border color. Default value is `None` which causes border style and border width to be ignored.

border_style

A string defining the line style of the field's border. See *Annot.border*. Default is "s" ("Solid") – a continuous line. Only the first character (upper or lower case) will be regarded when creating a widget.

border_width

A float defining the width of the border line. Default is 1.

border_dashes

A list/tuple of integers defining the dash properties of the border line. This is only meaningful if *border_style == "D"* and *border_color* is provided.

[1] If you intend to re-access a new or updated field (e.g. for making a pixmap), make sure to reload the page first. Either close and re-open the document, or load another page first, or simply do `page = doc.reload_page(page)`.

[2] Among other purposes, `Parent` objects are also used to facilitate multiple occurrences of a field (on the same or on different pages). The `Kids` array in this `Parent` object contains the cross references of all widgets that are "copies" of the same field. Whenever the field value of any "kid" widget is changed, all the other kids are immediately updated too. This is a very efficient way to handle multiple copies of the same field, e.g. for filling out forms. This simultaneous update only happens for `Widget.field value`. The new parameter `sync_flags` extends this to *Widget.field_flags*. This cannot be automated in the same way as for the field value to allow for more flexibility.

`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. If the name contains one or more colons "." the field is considered to be a child of a parent field. If the parent field does not exist, it will be created automatically. If the (full) name already exists anywhere in the PDF, the widget will become a new child of the existing field.

All widgets with the same name – whether or not a colon is part of it and independent of their position in the document – will share the same field value and field flags.

See also *the relationship between form fields and widgets*.

`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.

script_blur

- New in version 1.22.6

JavaScript text (unicode) to be performed on losing the focus of this field. `None` if not present.

script_focus

- New in version 1.22.6

JavaScript text (unicode) to be performed on focusing this field. `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`.

3. It is worthwhile to look at this manual with lots of information about Adobe's standard scripts for various field types. For example, if you want to add a text field representing a date, you may want to store the following scripts. They will ensure pattern-compatible date formats and display date pickers in supporting viewers:

```
widget.script_format = 'AFDate_FormatEx("mm/dd/yyyy");'
widget.script_stroke = 'AFDate_KeystrokeEx("mm/dd/yyyy");'
```

27.24.1 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.

27.24.2 Supported Widget Types

PyMuPDF supports the creation and update of most 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 now supports the creation and update of Radio Button Groups (RBGs). Adding a new radio button widget with the same (full) name as an existing one anywhere in the PDF will automatically create or extend an RBG.
- signature (`PDF_WIDGET_TYPE_SIGNATURE`) **read only** – no update or creation of signatures and no signing support.

27.24.3 The Relationship between Form Fields and Widgets

A form field is a logical object in the document's form field tree. It may have one or more widgets, which are the visual appearances of that field on one or more pages. A widget is therefore the page-bound representation of a form field.

The connection between a widget and its form field is established through the widget's `/Parent` entry, which references the form field. Conversely, the form field's `/Kids` array contains references to all widgets that visually represent that field.

Form fields themselves are not tied to any page; they exist solely in the document's AcroForm hierarchy. Widgets, however, are page annotations and are always associated with a specific page.

A single form field may be represented by multiple widgets. For example, a field may have one widget on page 1 and another on page 2 (or even multiple widgets on the same page). In this case, both widgets share the same form field as their parent. Changing the value of one widget automatically updates the value shown by all other widgets of that field.

27.24.4 Form Field Hierarchy

Form fields support hierarchical naming. A form field may have child fields, forming a logical structure. The hierarchy is defined by the `/Parent` entry of a child field, which references its parent field. The parent field's `/Kids` array contains references to all its child fields.

This hierarchy is reflected in the field's name. For example, a field named "Parent.Child" has a parent field "Parent" and a child field "Child". The parent may have additional children such as "Parent.Child2" or "Parent.Child3".

27.24.5 Automatic Hierarchy Creation via Dotted Names

Using dots (".") in a widget's `field_name` automatically creates or updates the corresponding form field hierarchy. For example:

- Creating a widget named "Person.Name" creates or updates the form field "Person" and adds the widget as a child of "Person".
- Creating a widget named "Person.Address" adds another child field under "Person".
- Creating a widget named "Person.MaritalStatus" adds yet another child field.

In all cases, the widget becomes the visual representation of the terminal field in the hierarchy.

This mechanism allows complex information structures to be expressed naturally within a PDF form.

This documentation covers all versions up to 1.28.2.

27.25 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
`add_bullet_list()`	Add a *ul* tag - bulleted list, context manager.
`add_codeblock()`	Add a *pre* tag, context manager.
`add_description_list()`	Add a *dl* tag, context manager.
`add_division()`	add a *div* tag (renamed from "section"), context manager.
`add_header()`	Add a header tag (one of *h1* to *h6*), context manager.
`add_horizontal_line()`	Add a *hr* tag.
`add_image()`	Add a *img* tag.
`add_link()`	Add a *a* tag.
`add_number_list()`	Add a *ol* tag, context manager.
`add_paragraph()`	Add a *p* tag.
`add_span()`	Add a *span* tag, context manager.
`add_subscript()`	Add subscript text(*sub* tag) - inline element, treated like text.
`add_superscript()`	Add subscript text (*sup* tag) - inline element, treated like text.
`add_code()`	Add code text (*code* tag) - inline element, treated like text.
`add_var()`	Add code text (*code* tag) - inline element, treated like text.
`add_samp()`	Add code text (*code* tag) - inline element, treated like text.
`add_kbd()`	Add code text (*code* tag) - inline element, treated like text.
`add_text()`	Add a text string. Line breaks `\n` are honored as *br* tags.
`append_child()`	Append a child node.
`clone()`	Make a copy if this node.
`create_element()`	Make a new node with a given tag name.
`create_text_node()`	Create direct text for the current node.
`find()`	Find a sub-node with given properties.
`find_next()`	Repeat previous "find" with the same criteria.
`insert_after()`	Insert an element after current node.
`insert_before()`	Insert an element before current node.
`remove()`	Remove this node.
`set_align()`	Set the alignment using a CSS style spec. Only works for block-level tags.
`set_attribute()`	Set an arbitrary key to some value (which may be empty).
`set_bgcolor()`	Set the background color. Only works for block-level tags.
`set_bold()`	Set bold on or off or to some string value.
`set_color()`	Set text color.
`set_columns()`	Set the number of columns. Argument may be any valid number or string.
`set_font()`	Set the font-family, e.g. "sans-serif".
`set_fontsize()`	Set the font size. Either a float or a valid HTML/CSS string.
`set_id()`	Set a *id*. A check for uniqueness is performed.
`set_italic()`	Set italic on or off or to some string value.
`set_leading()`	Set inter-block text distance (`-mupdf-leading`), only works on block-level nodes.
`set_lineheight()`	Set height of a line. Float like 1.5, which sets to `1.5 * fontsize`.
`set_margins()`	Set the margin(s), float or string with up to 4 values.
`set_pagebreak_after()`	Insert a page break after this node.
`set_pagebreak_before()`	Insert a page break before this node.
`set_properties()`	Set any or all desired properties in one call.
`add_style()`	Set (add) a "style" that is not supported by its own `set_` method.
`add_class()`	Set (add) a "class" attribute.
`set_text_indent()`	Set indentation for first textblock line. Only works for block-level nodes.

continues on next page

Table 5 – continued from previous page

Method / Attribute	Description
tagname	Either 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 the node is a text.
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.
root	The top node of the DOM, which hence has the tagname *html*.

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*)

Set 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*)

Set 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*)

Set 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*)

Set 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*)

Set 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 * fontsize`), or some valid line-height value.

`set_margins`(*value*)

Set 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 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 the first node with the given `tag`, attribute `att` and value `match`.

Parameters

- **tag** (*str*) – restrict search to this tag. May be `None` for unrestricted searches.

- **att** (*str*) – check this attribute. May be None.
- **match** (*str*) – the desired attribute value to match. May be None.

Return type
Xml.

Returns
None if nothing found, otherwise the first matching node.

find_next(*tag*, *att*, *match*)

Continue a previous *Xml.find()* (or *find_next()*) with the same values.

Return type
Xml.

Returns
None if none more found, otherwise the next matching node.

tagname

Either 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.

root

The top node of the DOM, which hence has the tagname *html*.

27.25.1 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>.

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.

27.25.2 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_txt("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")
```

This documentation covers all versions up to 1.28.2.

This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTYEIGHT

THE PYMUPDF4LLM API

property version

Prints the version of the library.

to_markdown(*doc: pymupdf.Document | str*, ***, *detect_bg_color: bool = True*, *dpi: int = 150*, *embed_images: bool = False*, *extract_words: bool = False*, *filename: str | None = None*, *fontsize_limit: float = 3*, *footer: bool = True*, *force_ocr: bool = False*, *force_text: bool = True*, *graphics_limit: int = None*, *hdr_info: Any = None*, *header: bool = True*, *ignore_alpha: bool = False*, *ignore_code: bool = False*, *ignore_graphics: bool = False*, *ignore_images: bool = False*, *image_format: str = 'png'*, *image_path: str = ''*, *image_size_limit: float = 0.05*, *margins: float | list = 0*, *ocr_dpi: int = 300*, *ocr_function: callable = None*, *ocr_language: str = 'eng'*, *page_chunks: bool = False*, *page_height: float = None*, *page_separators: bool = False*, *page_width: float = 612*, *pages: list | range | None = None*, *show_progress: bool = False*, *table_strategy: str = 'lines_strict'*, *use_glyphs: bool = False*, *use_ocr: bool = True*, *write_images: bool = False*) → str | list[dict]

Reads the pages of the file and outputs the text of its pages in format. How this should happen in detail can be influenced by a number of parameters. Please note that **support for building page chunks** from the text is supported.

Parameters

- **doc** (*Document*, *str*) – the file, to be specified either as a file path string, or as a *Document* (created via `pymupdf.open`). In order to use `pathlib.Path` specifications, Python file-like objects, documents in memory etc. you **must** use a *Document*.
- **detect_bg_color** (*bool*) – does a simple check for the general background color of the pages (default is `True`). If any text or vector has this color it will be ignored. May increase detection accuracy.
- **dpi** (*int*) – specify the desired image resolution in dots per inch. Relevant only if `write_images=True` or `embed_images=True`. Default value is 150.
- **embed_images** (*bool*) – like `write_images`, but images will be included in the markdown text as base64-encoded strings. Mutually exclusive with `write_images` and ignores `image_path`. This may drastically increase the size of your markdown text.
- **extract_words** (*bool*) – a value of `True` enforces `page_chunks=True` and adds key "words" to each page dictionary. Its value is a list of words as delivered by PyMuPDF's *Page* method `get_text("words")`. The sequence of the words in this list is the same as the extracted text.
- **filename** (*str*) – Overwrites or sets the desired image file name of written images. Useful when the document is provided as a memory object (which has no inherent file name).
- **fontsize_limit** (*float*) – limit the font size to consider for text extraction. If the font size is lower than what is set then the text won't be considered for extraction. Default is `3`, meaning only text with a font size >= `3` will be considered for extraction.

- **footer** (*bool*) – boolean to switch on/off page footer content. This parameter controls whether to include or omit footer text from all the document pages. Useful if the document has repetitive footer content which doesn't add any value to the overall extraction data. Default is `True` meaning that footer content will be considered.
- **force_ocr** (*bool*) – if `True`, OCR will be applied to all pages regardless of their content.

  This may be useful for documents which are known to be image-based and thus profit from OCR, but which do not meet the default criteria for applying OCR. Default is `False` meaning that OCR will only be applied to pages which meet the default criteria.

  > **Warning**
  >
  > Requires that either one of the default supported OCR engines is installed or `ocr_function` specifies a callable OCR function. Otherwise, an exception will be raised.

- **force_text** (*bool*) – generate text output even when overlapping images / graphics. This text then appears after the respective image.
- **graphics_limit** (*int*) – use this to limit dealing with excess amounts of vector graphics elements. Scientific documents, or pages simulating text via graphics commands may contain tens of thousands of these objects. As vector graphics are analyzed for multiple purposes, runtime may quickly become intolerable. With this parameter, all vector graphics will be ignored if their count exceeds the threshold.
- **hdr_info** – use this if you want to provide your own header detection logic. This may be a callable or an object having a method named *get_header_id*. It must accept a text span (a span dictionary as contained in *extractDICT()*) and a keyword parameter "page" (which is the owning *Page* object). It must return a string "" or up to 6 "#" characters followed by 1 space. If omitted (`None`), a full document scan will be performed to find the most popular font sizes and derive header levels based on them. To completely avoid this behavior specify `hdr_info=lambda s, page=None: ""` or `hdr_info=False`.
- **header** (*bool*) – boolean to switch on/off page header content. This parameter controls whether we want to include or omit the header content from all the document pages. Useful if the document has repetitive header content which doesn't add any value to the overall extraction data. Default is `True` meaning that header content will be considered.
- **ignore_alpha** (*bool*) – if `True` includes text even when completely transparent. Default is `False`: transparent text will be ignored which usually increases detection accuracy.
- **ignore_code** (*bool*) – if `True` then mono-spaced text lines do not receive special formatting. Code blocks will no longer be generated. This value is set to `True` if `extract_words=True` is used.
- **ignore_graphics** (*bool*) – (New in v.0.0.20) Disregard vector graphics on the page. This may help detecting text correctly when pages are very crowded (often the case for documents representing presentation slides). Also speeds up processing time. This automatically prevents table detection.
- **ignore_images** (*bool*) – (New in v.0.0.20) Disregard images on the page. This may help detecting text correctly when pages are very crowded (often the case for documents representing presentation slides). Also speeds up processing time.
- **image_format** (*str*) – specify the desired image format via its extension. Default is "png" (portable network graphics). Another popular format may be "jpg". Possible values are all *supported output formats*.

- **image_path** (*str*) – store images in this folder. Relevant if `write_images=True`. Default is the path of the script directory.
- **image_size_limit** (*float*) – this must be a `0 <= value < 1`. Images are ignored if `width / page.rect.width <= image_size_limit` or `height / page.rect.height <= image_size_limit`. For instance, the default value 0.05 means that to be considered for inclusion, an image's width and height must be larger than 5% of the page's width and height, respectively.
- **margins** (*float,list*) – a float or a sequence of 2 or 4 floats specifying page borders. Only objects inside the margins will be considered for output.
  - `margin=f` yields `(f, f, f, f)` for `(left, top, right, bottom)`.
  - `(top, bottom)` yields `(0, top, 0, bottom)`.
  - To always read full pages **(default)**, use `margins=0`.
- **ocr_dpi** (*int*) – specify the desired image resolution in dots per inch for applying OCR to the intermediate image of the page. Default value is 300. Only relevant if the page has been determined to profit from OCR (no or few text, most of the page covered by images or character-like vectors, etc.). Larger values do not usually increase the OCR precision. There also is a risk of over-sharpening the image which may decrease OCR precision. So the default value should probably be sufficiently high - in many cases you should see satisfactory results already with values of 150 or 200. Be aware that processing time and memory requirements grow quadratically with this value (an O(ocr_dpi^2) impact).
- **ocr_function** (*callable*) – if you want to provide your own OCR function, specify it here. If omitted (`None`), one of the available built-in OCR engines will be used.
- **ocr_language** (*str*) – specify the language to be used by the Tesseract OCR engine. Default is "eng" (English). Make sure that the respective language data files are installed. Remember to use correct Tesseract language codes. Multiple languages can be specified by concatenating the respective codes with a plus sign "+", for example "eng+deu" for English and German.
- **page_chunks** (*bool*) – if `True` the output will be a list of *Document.page_count* dictionaries (one per page). Each dictionary has the following structure:
  - **"metadata"** - a dictionary consisting of the document's metadata *Document.metadata*, enriched with additional keys **"file_path"** (the file name), **"page_count"** (number of pages in document), and **"page_number"** (1-based page number).
  - **"toc_items"** - a list of Table of Contents items pointing to this page. Each item of this list has the format `[lvl, title, pagenumber]`, where `lvl` is the hierarchy level, *title* a string and `pagenumber` as a 1-based page number.
  - **"tables"** - a list of tables on this page. Each item is a dictionary with keys "bbox", "row_count" and "col_count". Key "bbox" is a `pymupdf.Rect` in tuple format of the table's position on the page.
  - **"images"** - a list of images on the page. This a copy of page method *Page.get_image_info()*.
  - **"graphics"** - a list of vector graphics rectangles on the page. This is a list of boundary boxes of clustered vector graphics as delivered by method *Page.cluster_drawings()*.
  - **"text"** - page content as text.

- **"words"** - if `extract_words=True` was used. This is a list of tuples `(x0, y0, x1, y1, "wordstring", bno, lno, wno)` as delivered by `page.get_text("words")`. The **sequence** of these tuples however is the same as produced in the markdown text string and thus honors multi-column text. This is also true for text in tables: words are extracted in the sequence of table row cells.
- **"text"** - page content as text.
- **"page_boxes"** - a list of dictionaries representing the layout boundary boxes. Each dictionary has the following structure:

```
{
    "index": int,               # 0-based integer index of the␣
↪box in reading sequence
    "class": str,               # one of "text", "picture",
↪"table", etc.
    "bbox": [x0, y0, x1, y1],   # boundary box coordinates
    "pos": (start, stop),       # 0-based integers: bbox_text =␣
↪chunk["text"][start:stop]
}
```

See: *box classes*

- **page_height** (*float*) – specify a desired page height. For relevance see the `page_width` parameter. If using the default `None`, the document will appear as one large page with a width of `page_width`. Consequently in this case, no markdown page separators will occur (except the final one), respectively only one page chunk will be returned.
- **page_separators** (*bool*) – if `True` inserts a string `--- end of page=n ---` at the end of each page output. Intended for debugging purposes. The page number is 0-based. The separator string is wrapped with line breaks. Default is `False`.
- **page_width** (*float*) – specify a desired page width. This is ignored for documents with a fixed page width like PDF, XPS etc. **Reflowable** documents however, like e-books, office[2] or text files have no fixed page dimensions. They by default are assumed to have Letter format width (612) and an **unlimited** page height. This means that the **full document is treated as one large page.**
- **pages** (*list*) – optional, the pages to consider for output (caution: specify 0-based page numbers). If omitted (`None`) all pages are processed. Any Python sequence with integer items is accepted. The sequence is sorted and processed to only contain unique items.
- **show_progress** (*bool*) – Default is `False`. A value of `True` displays a progress bar as pages are being converted. Package tqdm is used if installed, otherwise the built-in text based progress bar is used.
- **table_strategy** (*str*) – see: *table detection strategy*. Default is `"lines_strict"` which ignores background colors. In some occasions, other strategies may be more successful, for example `"lines"` which uses all vector graphics objects for detection.
- **use_glyphs** (*bool*) – (New in v.0.0.19) Default is `False`. A value of `True` will use the glyph number of the characters instead of the character itself if the font does not store the Unicode value.
- **use_ocr** (*bool*) – use OCR capability to help analyse the page. This will OCR pages as determined by the default criteria.

[2] When using PyMuPDF-Pro, supported office documents are converted internally into a PDF-like format. Therefore, they **will have fixed page dimensions** and be no longer "reflowable". Consequently, the page width and page height specifications will be ignored as well in these cases.

- **write_images** (*bool*) – when encountering images or vector graphics, images will be created from the respective page area and stored in the specified folder. references will be generated pointing to these images. Any text contained in these areas will not be included in the text output (but appear as part of the images). Therefore, if for instance your document has text written on full page images, make sure to set this parameter to `False`.

  If using PyMuPDF Layout, boundary boxes that are classified as "picture" by the layout module will be treated as images - independent from the mixture of text, images or vector graphics they may be covering. If `force_text=True` is used, text will still be extracted from these areas and included in the output after the respective image reference.

Returns

Either a string of the combined text of all selected document pages, or a list of dictionaries if `page_chunks=True`.

to_text(*doc: pymupdf.Document | str*, *, ***kwargs*) → str

Reads the pages of the file and outputs the text of its pages in plain text () format.

Parameters

- **doc** (Document,*str*) – the file, to be specified either as a file path string, or as a *Document* (created via `pymupdf.open`). In order to use `pathlib.Path` specifications, Python file-like objects, documents in memory etc. you **must** use a *Document*.
- **use_ocr** (*bool*) – use OCR capability to help analyse the page.
- **ocr_language** (*str*) – specify the language to be used by the Tesseract OCR engine. Default is "eng" (English). Make sure that the respective language data files are installed. Remember to use correct Tesseract language codes. Multiple languages can be specified by concatenating the respective codes with a plus sign "+", for example "eng+deu" for English and German.
- **ocr_dpi** (*int*) – specify the desired image resolution in dots per inch for applying OCR to the intermediate image of the page. Default value is 400. Only relevant if the page has been determined to profit from OCR (no or few text, most of the page covered by images or character-like vectors, etc.). Large values may increase the OCR precision but increase memory requirements and processing time. There also is a risk of over-sharpening the image which may decrease OCR precision. So the default value should probably be sufficiently high.
- **header** (*bool*) – boolean to switch on/off page header content. This parameter controls whether to include or omit the header content from all the document pages. Useful if the document has repetitive header content which doesn't add any value to the overall extraction data. Default is `True` meaning that header content will be written.
- **footer** (*bool*) – boolean to switch on/off page footer content. This parameter controls whether to include or omit the footer content from all the document pages. Useful if the document has repetitive footer content which doesn't add any value to the overall extraction data. Default is `True` meaning that footer content will be written.
- **ignore_code** (*bool*) – if `True` then mono-spaced text lines do not receive special formatting. No blocks will be written and text lines will be written continuously.
- **pages** (*list*) – optional, the pages to consider for output (caution: specify 0-based page numbers). If omitted (`None`) all pages are processed. Any Python sequence with integer items is accepted. The sequence is sorted and processed to only contain unique items.
- **force_text** (*bool*) – generate text output also when overlapping images / graphics. This text then appears after the respective image reference. Images (i.e. "picture" areas) how-

ever will not be written to the text output but appear as a text line in the output like ==> `picture [width x height]` <==.

- **show_progress** (*bool*) – Default is `False`. A value of `True` displays a progress bar as pages are being converted. Package tqdm is used if installed, otherwise the built-in text based progress bar is used.
- **page_chunks** (*bool*) – if `True` the output will be a list of *Document.page_count* dictionaries (one per page). Each dictionary has the following structure:
  - **"metadata"** - a dictionary consisting of the document's metadata *Document.metadata*, enriched with additional keys **"file_path"** (the file name), **"page_count"** (number of pages in document), and **"page_number"** (1-based page number).
  - **"toc_items"** - a list of Table of Contents items pointing to this page. Each item of this list has the format `[lvl, title, pagenumber]`, where `lvl` is the hierarchy level, *title* a string and `pagenumber` as a 1-based page number.
  - **"tables"** - empty list.
  - **"images"** - empty list.
  - **"graphics"** - empty list.
  - **"words"** - empty list.
  - **"text"** - page content as plain text.
  - **"page_boxes"** - a list of dictionaries representing the layout boundary boxes. Each dictionary has the following structure:

```
{
    "index": int,               # 0-based integer index of the␣
↪box in reading sequence
    "class": str,               # one of "text", "picture",
↪"table", etc.
    "bbox": [x0, y0, x1, y1],   # boundary box coordinates
    "pos": (start, stop),       # 0-based integers: bbox_text =␣
↪chunk["text"][start:stop]
}
```

    See: *box classes*

to_json(*doc: pymupdf.Document | str*, *, ***kwargs*) → str

Parses the document and the specified pages and converts the result into a JSON formatted string.

Parameters

- **doc** (*Document*, *str*) – the file, to be specified either as a file path string, or as a *Document* (created via `pymupdf.open`). In order to use `pathlib.Path` specifications, Python file-like objects, documents in memory etc. you **must** use a *Document*.
- **use_ocr** (*bool*) – use OCR capability to help analyse the page.
- **ocr_language** (*str*) – specify the language to be used by the Tesseract OCR engine. Default is "eng" (English). Make sure that the respective language data files are installed. Remember to use correct Tesseract language codes. Multiple languages can be specified by concatenating the respective codes with a plus sign "+", for example "eng+deu" for English and German.
- **ocr_dpi** (*int*) – specify the desired image resolution in dots per inch for applying OCR to the intermediate image of the page. Default value is 400. Only relevant if the page has

been determined to profit from OCR (no or few text, most of the page covered by images or character-like vectors, etc.). Large values may increase the OCR precision but increase memory requirements and processing time. There also is a risk of over-sharpening the image which may decrease OCR precision. So the default value should probably be sufficiently high.

- **image_dpi** (*int*) – specify the desired image resolution in dots per inch. Default value is 150. Only relevant if one of the parameters `write_images=True` or `embed_images=True` is used.
- **image_format** (*str*) – specify the desired image format via its extension. Default is "png" (portable network graphics). Another popular format may be "jpg". Possible values are all *supported output formats*. Only relevant if one of the parameters `write_images=True` or `embed_images=True` is used.
- **image_path** (*str*) – store images in this folder. Relevant if `write_images=True`. Default is the path of the script directory. Page areas classified as "picture" will be written as image files to the specified location. The image file names will be of the format `{image_path}/{filename}-pagenumber-image_number.{image_format}`.
- **force_text** (*bool*) – generate text output for text that is written upon areas that are classified as "picture" by the layout module. This may be especially be useful when picture content is not stored.
- **show_progress** (*bool*) – display a progress bar during processing.
- **embed_images** (*bool*) – store image binaries for "picture" boundary boxes. Base64-encoded images are included in the JSON output. Ignores `image_path` if used. This may drastically increase the size of your JSON text.
- **write_images** (*bool*) – store image files "picture" boundary boxes. When encountering images, image files will be created from the respective page area and stored in the specified folder. Any text contained in these areas will still be included in the text output.
- **pages** (*list*) – optional, the pages to consider for output (caution: specify 0-based page numbers). If omitted (`None`) all pages are processed. Specify any valid Python sequence containing integers between `0` and `page_count - 1`.

Return type
str

See JSON Schema for the structure of the output JSON string.

Note

About box classes

If `page_chunks = True` the return objects for *to_markdown* & *to_text* contains a list of dictionaries representing the layout boundary boxes `page_boxes`, within that a key `class` indicates the type of box content therein.

The return object for *to_json* contains a similar key called `boxclass`.

The possible string values are for this `class` / `boxclass` key are:

```
text
picture
table
caption
title
section-header
page-header
page-footer
list-item
footnote
formula
```

use_layout(*yes: bool = True*)

Switch on/off the use of the *PyMuPDF Layout module*.

If `yes=True` (default), the layout module will be used for page analysis for optimal results. If `yes=False`, the layout module will not be used.

get_key_values(*doc: pymupdf.Document | str*) → list[dict]

Parse the document if it is a **Form PDF** and extract key-value pairs from all form fields (widgets).

Please note that this method is only relevant for PDF documents that contain widgets. Otherwise, an empty list will be returned.

The function is always available – independently of whether you are using the PyMuPDF Layout module or not.

Each dictionary item has the following structure:

```
{
    "field_name": str,      # the full name of the form field, components separated␣
↪by dots
    {
        "value": str,       # the field value as string
        "pages": list,      # list of 0-based page numbers where the field appears
    }
}
```

LlamaMarkdownReader(**args*, ***kwargs*)

Create a *pdf_markdown_reader.PDFMarkdownReader* using the LlamaIndex package. Please note that this package will **not automatically be installed** when installing **pymupdf4llm**.

For details on the possible arguments, please consult the LlamaIndex documentation[1].

Raises

`NotImplementedError`: Please install required LlamaIndex package.

Returns

a *pdf_markdown_reader.PDFMarkdownReader* and issues message "Successfully imported LlamaIndex". Please note that this method needs several seconds to execute. For details on using the markdown reader please see below.

---

class IdentifyHeaders

Note

__init__(*self*, *doc: pymupdf.Document | str*, ***, *pages: list | range | None = None*, *body_limit: float = 11*, *max_levels: int = 6*)

Create an object which maps text font sizes to the respective number of '#' characters which are used by Markdown syntax to indicate header levels. The object is created by scanning the document for font size "popularity". The most popular font size and all smaller sizes are used for body text. Larger font sizes are mapped to the respective header levels - which correspond to the HTML tags <h1> to <h6>.

[1] LlamaIndex documentation

All font sizes are rounded to integer values.

If more than 6 header levels would be required, then the largest number smaller than the <h6> font size is used for body text.

Please note that creating the object will read and inspect the text of the entire document - independently of reading the document again in the *to_markdown()* method subsequently. Method *to_markdown()* by default **will create this object** if you do not override its `hdr_info=None` parameter.

Parameters

- **doc** (*Document*, *str*) – the file, to be specified either as a file path string, or as a Document (created via `pymupdf.open`). In order to use `pathlib.Path` specifications, Python file-like objects, documents in memory etc. you **must** use a Document.
- **pages** (*list*) – optional, the pages to consider. If omitted all pages are processed.
- **body_limit** (*float*) – the default font size limit for body text. Only used when the document scan does not deliver valid information.
- **max_levels** (*int*) – the maximum number of header levels to be used. Valid values are in `range(1, 7)`. The default is 6, which corresponds to the HTML tags <h1> to <h6>. A smaller value will limit the number of generated header levels. For instance, a value of 3 will only generate header tags "#", "##" and "###". Body text will be assumed for all font sizes smaller than the one corresponding to "###".

get_header_id(*self*, *span: dict*, *page=None*) → str

Return appropriate markdown header prefix. This is either "" or a string of "#" characters followed by a space.

Given a text span from a "dict" extraction, determine the markdown header prefix string of 0 to n concatenated '#' characters.

Parameters

- **span** (*dict*) – a dictionary containing the text span information. This is the same dictionary as returned by `page.get_text("dict")`.
- **page** (*Page*) – the owning page object. This can be used when additional information needs to be extracted.

Returns

a string of "#" characters followed by a space.

header_id

A dictionary mapping (integer) font sizes to Markdown header strings like `{14: '# ', 12: '## '}`. The dictionary is created by the *IdentifyHeaders* constructor. The keys are the font sizes of the text spans in the document. The values are the respective header strings.

body_limit

An integer value indicating the font size limit for body text. This is computed as `min(header_id.keys()) - 1`. In the above example, body_limit would be 11.

How to limit header levels (example)

Limit the generated header levels to 3:

```
import pymupdf, pymupdf4llm

filename = "input.pdf"
doc = pymupdf.open(filename)  # use a Document for subsequent processing
my_headers = pymupdf4llm.IdentifyHeaders(doc, max_levels=3)  # generate header info
md_text = pymupdf4llm.to_markdown(doc, hdr_info=my_headers)
```

How to provide your own header logic (example 1)

Provide your own function which uses pre-determined, fixed font sizes:

```
import pymupdf, pymupdf4llm

filename = "input.pdf"
doc = pymupdf.open(filename)  # use a Document for subsequent processing

def my_headers(span, page=None):
    """
    Provide some custom header logic.
    This is a callable which accepts a text span and the page.
    Could be extended to check for other properties of the span, for
    instance the font name, text color and other attributes.
    """
    # header level is h1 if font size is larger than 14
    # header level is h2 if font size is larger than 10
    # otherwise it is body text
    if span["size"] > 14:
        return "# "
    elif span["size"] > 10:
        return "## "
    else:
        return ""

# this will *NOT* scan the document for font sizes!
md_text = pymupdf4llm.to_markdown(doc, hdr_info=my_headers)
```

How to provide your own header logic (example 2)

This user function uses the document's Table of Contents – under the assumption that the bookmark text is also present as a header line on the page (which certainly need not be the case!):

```
import pymupdf, pymupdf4llm

filename = "input.pdf"
doc = pymupdf.open(filename)  # use a Document for subsequent processing
TOC = doc.get_toc()  # use the table of contents for determining headers

def my_headers(span, page=None):
    """
    Provide some custom header logic (experimental!).
    This callable checks whether the span text matches any of the
    TOC titles on this page.
    If so, use TOC hierarchy level as header level.
    """
```

(continues on next page)

(continued from previous page)

```
    # TOC items on this page:
    toc = [t for t in TOC if t[-1] == page.number + 1]

    if not toc:  # no TOC items on this page
        return ""

    # look for a match in the TOC items
    for lvl, title, _ in toc:
        if span["text"].startswith(title):
            return "#" * lvl + " "
        if title.startswith(span["text"]):
            return "#" * lvl + " "

    return ""

# this will *NOT* scan the document for font sizes!
md_text = pymupdf4llm.to_markdown(doc, hdr_info=my_headers)
```

---

class TocHeaders

> **Note**

__init__(*self*, *doc: pymupdf.Document | str*)

Create an object which uses the document's Table of Contents (TOC) to determine header levels. Upon object creation, the table of contents is read via the *Document.get_toc()* method. The TOC data is then used to determine header levels in the *to_markdown()* method.

This is an alternative to *IdentifyHeaders*. Instead of running through the full document to identify font sizes, it uses the document's Table Of Contents (TOC) to identify headers on pages. Like *IdentifyHeaders*, this also is no guarantee to find headers, but for well-built Table of Contents, there is a good chance for more correctly identifying header lines on document pages than the font-size-based approach.

It also has the advantage of being much faster than the font-size-based approach, as it does not execute a full document scan or even access any of the document pages.

Examples where this approach works very well are the Adobe's files on PDF documentation.

Please note that this feature **does not read document pages** where the table of contents may exist as normal standard text. It only accesses data as provided by the *Document.get_toc()* method. It will not identify any headers for documents where the table of contents is not available as a collection of bookmarks.

get_header_id(*self*, *span: dict*, *page=None*) → str

Return appropriate markdown header prefix. This is either an empty string or a string of "#" characters followed by a space.

Given a text span from a "dict" extraction variant, determine the markdown header prefix string of 0 to n concatenated "#" characters.

Parameters

- **span** (*dict*) – a dictionary containing the text span information. This is the same dictionary as returned by `page.get_text("dict")`.
- **page** (Page) – the owning page object. This can be used when additional information needs to be extracted.

Returns
a string of "#" characters followed by a space.

How to use class TocHeaders

This is a version of previous **example 2** that uses *TocHeaders* for header identification:

```
import pymupdf, pymupdf4llm

filename = "input.pdf"

doc = pymupdf.open(filename)  # use a Document for subsequent processing
my_headers = pymupdf4llm.TocHeaders(doc)  # use the table of contents for determining␣
↪headers

# this will *NOT* scan the document for font sizes!
md_text = pymupdf4llm.to_markdown(doc, hdr_info=my_headers)
```

---

class pdf_markdown_reader.**PDFMarkdownReader**

load_data(*file_path: Path | str*, *extra_info: Dict | None = None*, ***load_kwargs: Any*) → List[LlamaIndexDocument]

This is the only method of the markdown reader you should currently use to extract markdown data. Please in any case ignore methods `aload_data()` and `lazy_load_data()`. Other methods like `use_doc_meta()` may or may not make sense. For more information, please consult the LlamaIndex documentation[Page 452, 1].

Under the hood the method will execute *to_markdown()*.

Returns
a list of `LlamaIndexDocument` documents - one for each page.

For a list of changes, please see file CHANGES.md.

This documentation covers all versions up to 1.28.2.

CHAPTER

TWENTYNINE

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.

29.1 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,** a bool or (for dot products) a float.
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 if "°" is a binary operator then the expression `a °= b` is always valid and the same as `a = a ° b`. Therefore, be careful and do **not** do `p1 *= p2` for two points, because thereafter "p1" is a **float**.

29.2 Unary Operations

Oper.	Result
bool(OBJ)	is false exactly if all components of OBJ are zero
abs(OBJ)	the rectangle area – equal to norm(OBJ) for the other types
norm(OBJ)	square root of the component squares (Euclidean norm)
+OBJ	new copy of OBJ
-OBJ	new copy of OBJ with negated components
~m	inverse of matrix "m", or the null matrix if not invertible

29.3 Binary Operations

These are expressions like `a ° b` where "°" is any of the operators `+, -, *, /`. Also binary operations are expressions of the form `a == b` and `b in a`.

If "b" is a number, then the respective operation is executed for each component of "a". Otherwise, if "b" is **not a number,** then the following happens:

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 and "m" is a *matrix_like*. *"a/m"* is treated as *"a*~m"* (see note below for non-invertible matrices). If "a" is a **point** or a **rectangle**, then *"a.transform(m)"* is executed. If "a" is a matrix, then matrix concatenation takes place.
a*b	returns the **vector dot product** for a point "a" and point-like "b".
a&b	**intersection rectangle:** "a" must be a rectangle and "b" *rect_like*. Delivers the **largest rectangle** contained in both operands.
a|b	**union rectangle:** "a" must be a rectangle, and "b" may be *point_like* or *rect_like*. Delivers the **smallest rectangle** containing both operands.
b in a	if "b" is a number, then `b in tuple(a)` is returned. If "b" is *point_like*, *rect_like* or *quad_like*, then "a" must be a rectangle, and `a.contains(b)` is returned.
a == b	`True` if *bool(a-b)* is `False` ("b" may be "a-like").

Note

Please note an important difference to usual arithmetic:

Matrix multiplication is **not commutative**, i.e. in general we have `m*n != 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 `pymupdf.Identity / m`. But if you formulate `pymupdf.Identity * ~m`, the result will be `pymupdf.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 `pymupdf.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 != r * (m1 * m2)
```

29.4 Some Examples

29.4.1 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)`":

```
>>> pymupdf.Rect(1, 2, 3, 4) + 5
pymupdf.Rect(6.0, 7.0, 8.0, 9.0)
>>> 3 in pymupdf.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 = pymupdf.Point(1, 2)
>>> p2 = pymupdf.Point(4711, 3141)
>>> mp = (p1 + p2) / 2
>>> mp
Point(2356.0, 1571.5)
>>>
```

Compute the **vector dot product** of two points. You can compute the **cosine of angles** and check orthogonality.

```
>>> p1 = pymupdf.Point(1, 0)
>>> p2 = pymupdf.Point(1, 1)
>>> dot = p1 * p2
>>> dot
1.0
```

```
>>> # compute the cosine of the angle between p1 and p2:
>>> cosine = dot / (abs(p1) * abs(p2))
>>> cosine  # cosine of 45 degrees
0.7071067811865475
```

```
>>> math.cos(mat.radians(45))  # verify:
0.7071067811865476
```

```
>>> # check orhogonality
>>> p3 = pymupdf.Point(0, 1)
>>> # p1 and p3 are orthogonal so, as expected:
```

(continues on next page)

(continued from previous page)

```
>>> p1 * p3
0.0
```

29.4.2 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:

```
>>> pymupdf.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 = pymupdf.Matrix(1, 2, 3, 4, 5, 6)
>>> n = pymupdf.Matrix(6, 5, 4, 3, 2, 1)
>>> p = pymupdf.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 = pymupdf.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:
```

(continues on next page)

(continued from previous page)

```
>>> p = pymupdf.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/pymupdf.py", line 869, in __truediv__
    raise ZeroDivisionError("matrix not invertible")
ZeroDivisionError: matrix not invertible
>>>
```

As a specialty, rectangles support additional binary operations:

- **intersection** – the common area of rectangle-likes, operator *"&"*
- **inclusion** – enlarge to include a point-like or rect-like, operator "|"
- **containment** check – whether a point-like or rect-like is inside

Here is an example for creating the smallest rectangle enclosing given points:

```
>>> # first define some point-likes
>>> points = []
>>> for i in range(10):
        for j in range(10):
            points.append((i, j))
>>>
>>> # now create a rectangle containing all these 100 points
>>> # start with an empty rectangle
>>> r = pymupdf.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
>>>
```

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTY

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.

30.1 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
`Annot.apn_bbox`	PDF only: bbox of the appearance object
`Annot.apn_matrix`	PDF only: the matrix of the appearance object
Page.is_wrapped	check whether contents wrapping is present
adobe_glyph_names()	list of glyph names defined in **Adobe Glyph List**
adobe_glyph_unicodes()	list of unicodes defined in **Adobe Glyph List**
Annot.clean_contents()	PDF only: clean the annot's *contents* object
`Annot.set_apn_bbox()`	PDF only: set the bbox of the appearance object
`Annot.set_apn_matrix()`	PDF only: set the matrix of the appearance object
ConversionHeader()	return header string for *get_text* methods
ConversionTrailer()	return trailer string for *get_text* methods
Document.del_xml_metadata()	PDF only: remove XML metadata
Document.get_char_widths()	PDF only: return a list of glyph widths of a font
Document.get_new_xref()	PDF only: create and return a new *xref* entry
Document.is_stream()	PDF only: check whether an *xref* is a stream object
Document.xml_metadata_xref()	PDF only: return XML metadata *xref* number
Document.xref_length()	PDF only: return length of *xref* table
EMPTY_IRECT()	return the (standard) empty / invalid rectangle
EMPTY_QUAD()	return the (standard) empty / invalid quad
EMPTY_RECT()	return the (standard) empty / invalid rectangle
get_pdf_now()	return the current timestamp in PDF format
get_pdf_str()	return PDF-compatible string
get_text_length()	return string length for a given font & *fontsize*
glyph_name_to_unicode()	return unicode from a glyph name
image_profile()	return a dictionary of basic image properties
INFINITE_IRECT()	return the (only existing) infinite rectangle
INFINITE_QUAD()	return the (only existing) infinite quad

continues on next page

Table 1 – continued from previous page

Function	Short Description
INFINITE_RECT()	return the (only existing) infinite rectangle
make_table()	split rectangle in sub-rectangles
Page.clean_contents()	PDF only: clean the page's *contents* objects
Page.get_bboxlog()	list of rectangles that envelop text, drawing or image objects
Page.get_contents()	PDF only: return a list of content *xref* numbers
Page.get_displaylist()	create the page's display list
Page.get_text_blocks()	extract text blocks as a Python list
Page.get_text_words()	extract text words as a Python list
Page.get_texttrace()	low-level text information
Page.read_contents()	PDF only: get complete, concatenated /Contents source
Page.run()	run a page through a device
Page.set_contents()	PDF only: set page's *contents* to some *xref*
Page.wrap_contents()	wrap contents with stacking commands
css_for_pymupdf_font()	create CSS source for a font in package pymupdf_fonts
paper_rect()	return rectangle for a known paper format
paper_size()	return width, height for a known paper format
paper_sizes()	dictionary of pre-defined paper formats
planish_line()	matrix to map a line to the x-axis
recover_char_quad()	compute the quad of a char ("rawdict")
recover_line_quad()	compute the quad of a subset of line spans
recover_quad()	compute the quad of a span ("dict", "rawdict")
recover_span_quad()	compute the quad of a subset of span characters
set_messages()	set destination of messages.
sRGB_to_pdf()	return PDF RGB color tuple from an sRGB integer
sRGB_to_rgb()	return (R, G, B) color tuple from an sRGB integer
unicode_to_glyph_name()	return glyph name from a unicode
get_tessdata()	locates the language support of the Tesseract-OCR installation
colors_pdf_dict()	return dict of color names.
colors_wx_list()	return list of color names.
fitz_fontdescriptors	dictionary of available supplement fonts
PYMUPDF_MESSAGE	destination of messages.
pdfcolor	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: *pymupdf.paper_size("A4")* returns *(595, 842)* and *pymupdf.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
pymupdf.Rect(0, 0, width, height) with *width, height=pymupdf.paper_size(s)*.

```
>>> import pymupdf
>>> pymupdf.paper_rect("letter-l")
pymupdf.Rect(0.0, 0.0, 792.0, 612.0)
>>>
```

---

set_messages(*, *text=None*, *fd=None*, *stream=None*, *path=None*, *path_append=None*, *pylogging=None*, *pylogging_logger=None*, *pylogging_level=None*, *pylogging_name=None*)

Sets destination of messages to a file descriptor, a file, an existing stream or Python's logging system.

Usually one would only set one arg, or one or more `pylogging*` args.

Parameters

- **text** (*str*) – A text specification of destination; for details see description of environmental variable *PYMUPDF_MESSAGE*.
- **fd** (*int*) – Write to file descriptor.
- **stream** – Write to existing stream, which must have methods `.write(text)` and `.flush()`.
- **path** (*str*) – Write to a file.
- **path_append** (*str*) – Append to a file.
- **pylogging** – Write to Python's `logging` system.
- **pylogging_logger** (*logging.Logger*) – Write to Python's `logging` system using specified Logger.
- **pylogging_level** (*int*) – Write to Python's `logging` system using specified level.
- **pylogging_name** (*str*) – Write to Python's `logging` system using specified logger name. Only used if `pylogging_logger` is `None`. Default is `pymupdf`.

If any `pylogging*` arg is not `None`, we write to Python's logging system.

---

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 <= item <= 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

Returns
the 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

Returns
the glyph name. E.g. `pymupdf.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 = pymupdf.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 `pymupdf.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 = pymupdf.Archive()
CSS = pymupdf.css_for_pymupdf_font("notos", name="sans-serif",␣
↪archive=arch)
story = pymupdf.Story(user_css=CSS, archive=arch)
```

make_table(*rect*, *cols=1*, *rows=1*)

New in v1.17.4

Convenience function to split a rectangle into sub-rectangles of equal size. 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 = pymupdf.make_table(rect, cols=4, rows=3)`:

cell[0][0]	cell[0][1]	cell[0][2]	cell[0][3]
cell[1][0]	cell[1][1]	cell[1][2]	cell[1][3]
cell[2][0]	cell[2][1]	cell[2][2]	cell[2][3]

---

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.

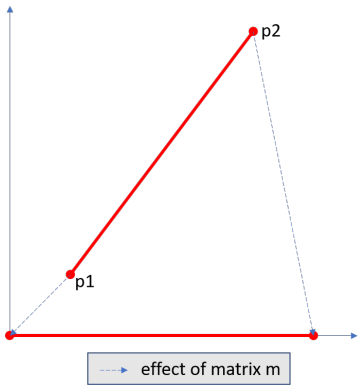
Return type

Matrix

Returns

a matrix which combines a rotation and a translation:

```
>>> p1 = pymupdf.Point(1, 1)
>>> p2 = pymupdf.Point(4, 5)
>>> abs(p2 - p1)  # distance of points
5.0
>>> m = pymupdf.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]: pymupdf.fitz_fontdescriptors.keys()
Out[2]: dict_keys(['figbo', 'figo', 'figbi', 'figit', 'fimbo', 'fimo',
'spacembo', 'spacembi', 'spacemit', 'spacemo', 'math', 'music', 'symbol1',
'symbol2'])
In [3]: pymupdf.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 = pymupdf.Font("fimo")` – just like you can do it with the builtin fonts "Helvetica" and friends.

PYMUPDF_MESSAGE

If in `os.environ` when is imported, sets destination of messages. Otherwise messages are sent to `sys.stdout`.

- If the value starts with `fd:`, the remaining text should be an integer file descriptor to which messages are written.
  - For example `PYMUPDF_MESSAGE=fd:2` will send messages to stderr.
- If the value starts with `path:`, the remaining text is the path of a file to which messages are written. If the file already exists, it is truncated.

- If the value starts with `path+:`, the remaining text is the path of file to which messages are written. If the file already exists, we append output.
- If the value starts with `logging:`, messages are written to Python's logging system. The remaining text can contain comma-separated name=value items:
  - `level`= sets the logging level.
  - `name`= sets the logger name (default is `pymupdf`).

  Other items are ignored.
- Other prefixes will cause an error.

Also see *set_messages()*.

---

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 `pymupdf.pdfcolor.keys()`.

Examples:

- `pymupdf.pdfcolor["red"] = (1.0, 0.0, 0.0)`
- `pymupdf.pdfcolor["skyblue"] = (0.5294117647058824, 0.807843137254902, 0.9215686274509803)`
- `pymupdf.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 font name. Must be one of either the *PDF Base 14 Fonts* or the CJK fonts, identified by their "reserved" fontnames (see table in *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([pymupdf.get_text_length(c) for c in text]) > pymupdf.get_text_length(text)*. So either (1) do the same, or (2) use something like *pymupdf.get_text_length(text + "'")* for your calculation.

---

get_pdf_str(*text*)

Make a PDF-compatible string: if the text contains code points *ord(c) > 255*, then it will be converted to UTF-16BE with BOM as a hexadecimal character string enclosed in "<>" brackets like <*feff*...>. Otherwise, it will return the string enclosed in (round) brackets, replacing any characters outside the ASCII range with some special code. Also, every "(", ")" or backslash is escaped with 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.
- Changed in v1.22.5: always return `None` in error cases instead of an empty dictionary.

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*) – either an image in memory or an **opened** file. An image in memory may be any of the formats `bytes`, `bytearray` or `io.BytesIO`.

Return type

dict

Returns

No exception is ever raised. In case of an error, `None` is returned. Otherwise, there are the following items:

```
In [2]: pymupdf.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. You can however still work with such images in PyMuPDF, e.g. by using *Document.extract_image()* or cre-

ate 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: `pymupdf.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**(*layers=False*)

- New in v1.19.0
- Changed in v1.22.0: optionally also return the OCG name applicable to the boundary box.

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. If `layers=True`, there is a third item containing the OCG name or `None`: `(type, (x0, y0, x1, y1), None)`.

- `"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 can be used to detect such situations. An item's index in this list equals the value of a `"seqno"` in dictionaries as 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)`.

- Changed in v1.22.0: add new dictionary key "layer".

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)
   'layer': None,                      # name of Optional Content Group (9)
   '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 <= f <= 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 <= opacity <= 1`, 0 is invisible text, 1 (100%) is intransparent. Depending on `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.
9. *(New in v1.22.0)* The name of the Optional Content Group (OCG) if applicable or `None`.

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 0xFFFD (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 *recover_span_quad()* as explained in *Structure of Dictionary Outputs*. Use either `None` or `span["dir"]` for the writing direction.
- **Changed in v1.21.1:** If applicable, the name of the OCG is shown in `"layer"`.

Page.**wrap_contents**()

Ensures that the page's so-called graphics state is balanced and new content can be inserted correctly.

In versions 1.24.1+ of PyMuPDF the method was improved and is being executed automatically as required, so you should no longer need to concern yourself with it.

We discourage using *Page.clean_contents()* to achieve this.

Page.**is_wrapped**

Indicate whether the page's so-called graphic state is balanced. If `False`, *Page.wrap_contents()* should be executed if new content is inserted (only relevant in `overlay=True` mode). In newer versions (1.24.1+), this check and corresponding adjustments are automatically executed – you therefore should not be concerned about this anymore.

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*, *delimiters=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 no more than one entry. 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 snychronized. 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"*.
>
> Do not any longer use this method to ensure correct insertions on PDF pages. Since PyMuPDF version 1.24.2 this is taken care of automatically.

`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(f"max. code point found: {ord(max(text))},␣
↪increase limit")
```

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 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.).

---

get_tessdata(*tessdata=None*)

Detect Tesseract language support folder.

This function is used to enable OCR via Tesseract even if the language support folder is not specified directly or in environment variable TESSDATA_PREFIX.

- If <tessdata> is set we return it directly.

- Otherwise we return `os.environ['TESSDATA_PREFIX']` if set.
- Otherwise we search for a Tesseract installation and return its language support folder.
- Otherwise we raise an exception.

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.

---

colors_pdf_dict()

Returns a dict mapping lower-case color name to `(red, green, blue)` tuple, and `red`, `green`, `blue` are floats in range 0..1.

colors_wx_list()

Returns a list of `(colorname, red, green, blue)` tuples, where `colorname` is upper case and `red`, `green`, `blue` are integers in range 0..255.

This documentation covers all versions up to 1.28.2.

30.2 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 *Font Properties*. To set these options use something like `flags=0 | TEXT_PRESERVE_LIGATURES | ...`.

This documentation covers all versions up to 1.28.2.

30.3 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.

30.3.1 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.

30.3.2 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()*.

30.3.3 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
```

30.3.4 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
```

30.3.5 Further Performance improvements

Pixmap

As explained in the *Page* chapter:

If you do not need transparency set *alpha = 0* when creating pixmaps. This will save 25% memory (if RGB, the most common case) and possibly 5% execution time (depending on the GUI software).

TextPage

If you do not need images extracted alongside the text of a page, you can set the following option:

```
>>> flags = pymupdf.TEXT_PRESERVE_LIGATURES | pymupdf.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 = pymupdf.TEXT_PRESERVE_LIGATURES | pymupdf.TEXT_PRESERVE_WHITESPACE | pymupdf.
↪TEXT_PRESERVE_IMAGES
```

This documentation covers all versions up to 1.28.2.

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTYONE

GLOSSARY

coordinate

This is an essential general mathematical / geometrical term for understanding this documentation. Please see this section for a more detailed discussion: *Coordinates*.

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 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 y coordinates such that the **top** border 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
         /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)
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*.

fontsize

When referring to font size this metric is measured in points where 1 inch = 72 points.

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. If it finds no 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.

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTYTWO

CONSTANTS AND ENUMERATIONS

Constants and enumerations of *MuPDF* as implemented by . Each of the following values is accessible as `pymupdf.value`.

32.1 Constants

`Base14_Fonts`

Predefined Python list of valid *PDF Base 14 Fonts*.

Type
list

`csRGB`

Predefined RGB colorspace *pymupdf.Colorspace(pymupdf.CS_RGB)*.

Type
Colorspace

`csGRAY`

Predefined GRAY colorspace *pymupdf.Colorspace(pymupdf.CS_GRAY)*.

Type
Colorspace

`csCMYK`

Predefined CMYK colorspace *pymupdf.Colorspace(pymupdf.CS_CMYK)*.

Type
Colorspace

`CS_RGB`

1 – Type of *Colorspace* is RGBA

Type
int

`CS_GRAY`

2 – Type of *Colorspace* is GRAY

Type
int

`CS_CMYK`

3 – Type of *Colorspace* is CMYK

Type
int

mupdf_version
'x.xx.x' – MuPDF version that is being used by PyMuPDF.

Type
string

mupdf_version_tuple
MuPDF version as a tuple of integers, `(major, minor, patch)`.

Type
tuple

pymupdf_version
'x.xx.x' – PyMuPDF version.

Type
string

pymupdf_version_tuple
PyMuPDF version as a tuple of integers, `(major, minor, patch)`.

Type
tuple

pymupdf_date
Disabled (set to None) in 1.26.1.

version
(pymupdf_version, mupdf_version, timestamp) – combined version information where `timestamp` is the generation point in time formatted as "YYYYMMDDhhmmss".

Type
tuple

VersionBind
Legacy equivalent to *mupdf_version*.

VersionFitz
Legacy equivalent to *pymupdf_version*.

VersionDate
Disabled (set to None) in 1.26.1.

32.2 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

32.3 PDF Optional Content Codes

Code	Meaning
PDF_OC_ON	Set an OCG to ON temporarily
PDF_OC_TOGGLE	Toggle OCG status temporarily
PDF_OC_OFF	Set an OCG to OFF temporarily

32.4 PDF encryption method codes

Code	Meaning
PDF_ENCRYPT_KEEP	do not change
PDF_ENCRYPT_NONE	remove any encryption
PDF_ENCRYPT_RC4_40	RC4 40 bit
PDF_ENCRYPT_RC4_128	RC4 128 bit
PDF_ENCRYPT_AES_128	*Advanced Encryption Standard* 128 bit
PDF_ENCRYPT_AES_256	*Advanced Encryption Standard* 256 bit
PDF_ENCRYPT_UNKNOWN	unknown

32.5 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. *PDF Base 14 Fonts*, Type 3 fonts and others

32.6 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.

32.7 Font Properties

Please note that the following bits are derived from what a font has to say about its properties. It may not be (and quite often is not) correct.

TEXT_FONT_SUPERSCRIPT

1 – the character or span is a superscript. This property is computed by MuPDF and not part of any font information.

TEXT_FONT_ITALIC

2 – the font is italic.

TEXT_FONT_SERIFED

4 – the font is serifed.

TEXT_FONT_MONOSPACED

8 – the font is mono-spaced.

TEXT_FONT_BOLD

16 – the font is bold.

32.8 Text Extraction Flags

Option bits controlling the amount of data, that are parsed into a *TextPage*.

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_CLIP, TEXT_MEDIABOX_CLIP

64 – Characters entirely outside a page's **mediabox** or contained in other "clipped" areas will be ignored. This is default in PyMuPDF. (TEXT_MEDIABOX_CLIP is an old alias.)

TEXT_USE_CID_FOR_UNKNOWN_UNICODE

128 – Use raw character codes instead of U+FFFD. This is the default for **text extraction** in PyMuPDF. If you **want to detect** when encoding information is missing or uncertain, toggle this flag and scan for the presence of U+FFFD (= `chr(0xfffd)`) code points in the resulting text.

TEXT_COLLECT_STRUCTURE

256 – Extract or generate the *Document* structure. Detail documentation pending.

TEXT_ACCURATE_BBOXES

512 – Ignore metric values of all fonts when computing character boundary boxes – most prominently the ascender and descender values. Instead, follow the drawing commands of each character's glyph and compute their rectangle hull as the bbox. This is the smallest rectangle wrapping all points used for drawing the visual appearance - see the *Shape* class for understanding the background. This will especially result in individual character heights. For instance a (white) space will have a **bbox of zero height** (because nothing is drawn) – in contrast to the non-zero boundary box generated when using font metrics. This option may be useful to cope with failures of getting meaningful boundary boxes, even for fonts containing errors. Its use will slow down text extraction somewhat because of the incurred computational effort.

Note that this has no effect by default - one must also disable the global quad corrections setting with `pymupdf.TOOLS.unset_quad_corrections(True)`.

TEXT_COLLECT_VECTORS

1024 – Collect vector drawings into the *TextPage*. These are stored as blocks alongside text and image blocks, depending on other extraction flags. See *TextPage.extractBLOCKS()* and *TextPage.extractDICT()* for details. Beyond these two methods, vector graphics extraction is also available for *TextPage.extractJSON()*, *TextPage.extractRAWDICT()*, *TextPage.extractRAWJSON()* and *TextPage.extractXML()*.

TEXT_IGNORE_ACTUALTEXT

2048 – Ignore built-in differences between text appearing in e.g. PDF viewers versus text stored in the PDF. See *Adobe PDF References*, page 615 for background. If set, the **stored** ("replacement" text) is ignored in favor of the displayed text.

TEXT_SEGMENT

4096 – Attempt to segment page into different regions. Detail documentation pending.

TEXT_COLLECT_STYLES

32768 – Request collecting text **decoration** properties. This includes text underlining and strikeout. In contrast to public awareness, these are not font properties, but are drawn separately as vector graphics or annotations on top of the text. In addition, the flag bit will also cause MuPDF to detect "fake bold" text. In many cases, Document creators **simulate bold** text by printing the same text multiple times with slight offsets. If this flag is set, such text will be marked as bold in the resulting text spans.

TEXT_LAZY_VECTORS

1048576 – Delay vector blocks in the extraction slightly to avoid breaking what would otherwise be continuous lines of text.

TEXT_FUZZY_VECTORS

2097152 – If this option is set, we 'fuzzily' collect rectangular vectors of the same colour together. This enables

us to spot where 'pixels' or 'slices' of vectors are used to create the appearance of characters on the page without exploding the storage and processing time requirements.

The following constants represent the default combinations of the above for text extraction and searching:

TEXTFLAGS_TEXT

TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE | TEXT_MEDIABOX_CLIP | TEXT_USE_CID_FOR_UNKNOWN_UNICODE

TEXTFLAGS_WORDS

TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE | TEXT_MEDIABOX_CLIP | TEXT_USE_CID_FOR_UNKNOWN_UNICODE

TEXTFLAGS_BLOCKS

TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE | TEXT_MEDIABOX_CLIP | TEXT_USE_CID_FOR_UNKNOWN_UNICODE

TEXTFLAGS_DICT

TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE | TEXT_MEDIABOX_CLIP | TEXT_PRESERVE_IMAGES | TEXT_USE_CID_FOR_UNKNOWN_UNICODE

TEXTFLAGS_RAWDICT

TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE | TEXT_MEDIABOX_CLIP | TEXT_PRESERVE_IMAGES | TEXT_USE_CID_FOR_UNKNOWN_UNICODE

TEXTFLAGS_HTML

TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE | TEXT_MEDIABOX_CLIP | TEXT_PRESERVE_IMAGES | TEXT_USE_CID_FOR_UNKNOWN_UNICODE

TEXTFLAGS_XHTML

TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE | TEXT_MEDIABOX_CLIP | TEXT_PRESERVE_IMAGES | TEXT_USE_CID_FOR_UNKNOWN_UNICODE

TEXTFLAGS_XML

TEXT_PRESERVE_LIGATURES | TEXT_PRESERVE_WHITESPACE | TEXT_MEDIABOX_CLIP | TEXT_USE_CID_FOR_UNKNOWN_UNICODE

TEXTFLAGS_SEARCH

TEXT_PRESERVE_WHITESPACE | TEXT_MEDIABOX_CLIP | TEXT_DEHYPHENATE

32.9 Link Destination Kinds

Possible values of *linkDest.kind* (link destination kind).

LINK_NONE

0 – No destination. Indicates a dummy link.

Type
int

LINK_GOTO

1 – Points to a place in this document.

Type
int

LINK_URI

2 – Points to a URI – typically a resource specified with internet syntax.

- PyMuPDF treats any external link that contains a colon and does not start with `file:`, as *LINK_URI*.

Type
int

LINK_LAUNCH

3 – Launch (open) another file (of any "executable" type).

- treats any external link that starts with `file:` or doesn't contain a colon, as *LINK_LAUNCH*.

Type
int

LINK_NAMED

4 – points to a named location.

Type
int

LINK_GOTOR

5 – Points to a place in another PDF document.

Type
int

32.10 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

Type
bool

LINK_FLAG_T_VALID

2 (bit 1) Top left y value is valid

Type
bool

LINK_FLAG_R_VALID

4 (bit 2) Bottom right x value is valid

Type
bool

LINK_FLAG_B_VALID

8 (bit 3) Bottom right y value is valid

Type
bool

LINK_FLAG_FIT_H

16 (bit 4) Horizontal fit

Type
bool

LINK_FLAG_FIT_V

32 (bit 5) Vertical fit

Type
bool

LINK_FLAG_R_IS_ZOOM

64 (bit 6) Bottom right x is a zoom figure

Type
bool

32.11 Annotation Related Constants

See chapter 8.4.5, pp. 615 of the *Adobe PDF References* for details.

32.11.1 Annotation Types

These identifiers also cover **links** and **widgets**: the PDF specification technically handles them all in the same way, whereas (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
```

(continues on next page)

(continued from previous page)

```
PDF_ANNOT_WATERMARK 25
PDF_ANNOT_3D 26
PDF_ANNOT_PROJECTION 27
PDF_ANNOT_UNKNOWN -1
```

32.11.2 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)
```

32.11.3 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
```

32.12 Widget Constants

32.12.1 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
```

32.12.2 Text Widget Subtypes (*text_format*)

```
PDF_WIDGET_TX_FORMAT_NONE 0
PDF_WIDGET_TX_FORMAT_NUMBER 1
PDF_WIDGET_TX_FORMAT_SPECIAL 2
```

(continues on next page)

(continued from previous page)

```
PDF_WIDGET_TX_FORMAT_DATE 3
PDF_WIDGET_TX_FORMAT_TIME 4
```

32.12.3 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
```

32.13 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"
PDF_BM_Difference "Difference"
PDF_BM_Exclusion "Exclusion"
PDF_BM_HardLight "HardLight"
PDF_BM_Hue "Hue"
PDF_BM_Lighten "Lighten"
PDF_BM_Luminosity "Luminosity"
```

(continues on next page)

(continued from previous page)

```
PDF_BM_Multiply "Multiply"
PDF_BM_Normal "Normal"
PDF_BM_Overlay "Overlay"
PDF_BM_Saturation "Saturation"
PDF_BM_Screen "Screen"
PDF_BM_SoftLight "Softlight"
```

32.14 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
```

32.15 Pixmap Transformations

The following constants are defined inside the `Pixmap` class. They are used to select one of the following transformations.

```
Pixmap.ROTATE_90 1
Pixmap.ROTATE_270 2
Pixmap.ROTATE_180 3
Pixmap.FLIP_LEFT_RIGHT 4
Pixmap.FLIP_TOP_BOTTOM 5
```

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTYTHREE

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 memorizable names. Among them are not only standard names like "green" or "blue", but also "turquoise", "skyblue", and 100 (not only 50 ...) shades of "gray", etc.

We have taken the liberty to copy this database (a list of tuples) modified into PyMuPDF and make its colors available as PDF compatible float triples: for wxPython's *("WHITE", 255, 255, 255)* we return *(1, 1, 1)*, which can be directly used in *color* and *fill* parameters. We also accept any mixed case of "wHiTe" to find a color.

33.1 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 pymupdf.utils import getColor
>>> getColor("aliceblue")
(0.9411764705882353, 0.9725490196078431, 1.0)
>>> #
>>> # to get a list of all existing names
>>> from pymupdf.utils import getColorList
>>> cl = getColorList()
>>> cl
['ALICEBLUE', 'ANTIQUEWHITE', 'ANTIQUEWHITE1', 'ANTIQUEWHITE2', 'ANTIQUEWHITE3',
'ANTIQUEWHITE4', 'AQUAMARINE', 'AQUAMARINE1'] ...
>>> #
>>> # to see the full integer color coding
>>> from pymupdf.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)] ...
```

33.2 Printing the Color Database

If you want to actually see how the many available colors look like, use scripts print by RGB or print by HSV 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.

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTYFOUR

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)?

34.1 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>
    <image block>
        <img>
```

A **text page** consists of blocks (= roughly paragraphs).

A **block** consists of either lines and their characters, or an image.

A **line** consists of spans.

A **span** consists of adjacent characters with identical font properties: name, size, flags and color.

34.2 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)`.

34.3 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)]
```

34.4 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)
```

34.5 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>
```

34.6 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 successful

while True:
    pos0 = otext.find("font-family:", pos1)    # start of a font spec
    if pos0 < 0:                               # none found - we are done
        break
    pos1 = otext.find(";", pos0)               # end of font spec
    test = otext[pos0 : pos1]                  # complete font spec string
    testn = ""                                 # the new font spec string
    if test.endswith(",serif"):                # font with serifs?
        testn = font_serif                     # use Times instead
    elif test.endswith(",sans-serif"):         # sans serifs font?
        testn = font_sans                      # use Helvetica
```

(continues on next page)

(continued from previous page)

```
    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!")
```

34.7 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)
            })
        }]
    }]
}
```

34.8 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": "."
}],
```

34.9 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"/>
<char quad="57.337 88.175 63.453004 88.175 57.337 103.289 63.453004 103.289" x="57.337"
```

(continues on next page)

(continued from previous page)

```
y="100" color="#000000" c="o"/>
<char quad="63.453004 88.175 72.616008 88.175 63.453004 103.289 72.616008 103.289" x="63.
↪453004"
y="100" color="#000000" c="m"/>
<char quad="72.616008 88.175 78.732 88.175 72.616008 103.289 78.732 103.289" x="72.616008
↪"
y="100" color="#000000" c="e"/>
<char quad="78.732 88.175 81.79 88.175 78.732 103.289 81.79 103.289" x="78.732"
y="100" color="#000000" c=" "/>

... deleted ...

<char quad="163.11298 88.175 166.17098 88.175 163.11298 103.289 166.17098 103.289" x=
↪"163.11298"
y="100" color="#000000" c="."/>
</font>
</line>
</block>
</page>
```

Note

We have successfully tested lxml to interpret this output.

34.10 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>
```

34.11 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 *Font Properties*.
- 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	0
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
use CID instead of U+FFFD	1	1	1	1	1	1	1	1	0

- **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 !
Mo r e  t e x t
i s  f o l l o w i n g
i n  E n g l i s h
. . .  l e t ' s  s e e
w h a t  h a p p e n s .
>>> print(page.get_text("text", flags=pymupdf.TEXT_INHIBIT_SPACES))
Hallo!
More text
is following
in English
... let's see
what happens.
>>>
```

34.12 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 where excluded, less than 50% of that time (77 seconds) were needed.

To begin with, all methods are **very fast** in relation to other products out there in the market. In terms of processing speed, we are not aware of a faster (free) tool. Even the most detailed method, RAWDICT, processes all 1'310 pages of the *Adobe PDF 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.

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTYFIVE

APPENDIX 2: CONSIDERATIONS ON EMBEDDED FILES

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

35.1 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.

35.2 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.

35.3 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 registered in a PDF, `Document.embfile_names()`.

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTYSIX

APPENDIX 3: ASSORTED TECHNICAL INFORMATION

This section deals with various technical topics, that are not necessarily related to each other.

36.1 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 = pymupdf.Rect(0, 0, width, height)`
   - define the "shrink matrix" `shrink = pymupdf.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 = pymupdf.Rect(0, 0, 1, 1)`.
3. Using the image **transformation matrix** "transform", the following steps will compute the bbox:

```
imgrect = pymupdf.Rect(0, 0, width, height)
shrink = pymupdf.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 = pymupdf.Matrix(1 / 439, 0, 0, 1 / 501, 0, 0)
>>> imgrect = pymupdf.Rect(0, 0, 439, 501)
>>> #-------------------------------------------------
>>> # determine image bbox and transformation matrix:
```

(continues on next page)

(continued from previous page)

```
>>> #-----------------------------------------------
>>> 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, -0.3992015719413757, 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 pymupdf.Matrix(-90) * 0.4
>>> #-----------------------------------------------
```

36.2 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 amd 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]: pymupdf.Base14_fontdict
Out[2]:
{'courier': 'Courier',
'courier-oblique': 'Courier-Oblique',
'courier-bold': 'Courier-Bold',
'courier-boldoblique': 'Courier-BoldOblique',
'helvetica': 'Helvetica',
'helvetica-oblique': 'Helvetica-Oblique',
'helvetica-bold': 'Helvetica-Bold',
'helvetica-boldoblique': 'Helvetica-BoldOblique',
'times-roman': 'Times-Roman',
'times-italic': 'Times-Italic',
'times-bold': 'Times-Bold',
'times-bolditalic': 'Times-BoldItalic',
'symbol': 'Symbol',
'zapfdingbats': 'ZapfDingbats',
'helv': 'Helvetica',
'heit': 'Helvetica-Oblique',
'hebo': 'Helvetica-Bold',
'hebi': 'Helvetica-BoldOblique',
'cour': 'Courier',
'coit': 'Courier-Oblique',
'cobo': 'Courier-Bold',
'cobi': 'Courier-BoldOblique',
'tiro': 'Times-Roman',
```

(continues on next page)

(continued from previous page)

```
'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()*.

36.3 Adobe PDF References

This PDF Reference manual published by Adobe is frequently quoted throughout this documentation. It can be viewed and downloaded from opensource.adobe.com.

36.4 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:

- `pymupdf.Point(s)`
- `pymupdf.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 pymupdf, array, numpy as np
>>> m = pymupdf.Matrix(1, 2, 3, 4, 5, 6)
>>>
>>> list(m)
```

(continues on next page)

(continued from previous page)

```
[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.

36.5 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: *pymupdf.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 *pymupdf.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()*, *delePage()*, *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 = pymupdf.open("some.pdf")
>>> page = doc[n]
>>> annot = page.first_annot
>>> page.rect
pymupdf.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.

36.6 Design of Method Page.show_pdf_page()

36.6.1 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.

36.6.2 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()*, the following things happen:

1. The *resources* and *contents* objects of source page in source document are copied over to the target 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.
   c. /Resources equals that of the source page. This involves a "deep-copy" of hierarchically nested other objects (including fonts, images, etc.). The complexity involved here is covered by MuPDF's grafting[1] 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 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.

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 Form XObject is only executed once per shown source page. Subsequent displays of the same source page will skip this step and only create "pointer" Form XObjects (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 Form XObject via the fixed name `fullpage`.
   d. The stream of this object contains exactly one fixed statement: `/fullpage Do`.
   e. If the method's `"oc"` argument is given, its value is assigned to this Form XObject as `/OC`.
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* (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`.

This design approach ensures that:

1. The (potentially large) source page is only copied once to the target PDF. Only small "pointer" Form XObjects objects are created per each target page to show the source page.
2. Each referring target page can have its own `"oc"` parameter to control the source page's visibility individually.

36.7 Diagnostics

36.7.1 messages

has a Message system for showing text diagnostics.

By default messages are written to `sys.stdout`. This can be controlled in two ways:

- Set environment variable *PYMUPDF_MESSAGE* before is imported.
- Call *set_messages()*:

36.7.2 errors and warnings

generates text errors and warnings.

- These errors and warnings are appended to an internal list, accessible with *Tools.mupdf_warnings()*. Also see *Tools.reset_mupdf_warnings()*.
- By default these errors and warnings are also sent to the message system.
  - This can be controlled with *mupdf_display_errors()* and *mupdf_display_warnings()*.
  - These messages are prefixed with `MuPDF error:` and `MuPDF warning:` respectively.

Some errors may lead to Python exceptions.

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 `pymupdf.open` the document had to be repaired:

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

Example output for an **unrecoverable error**:

```
>>> import pymupdf
>>> doc = pymupdf.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 = pymupdf.open("does-not-exist.pdf")
  File "C:\Users\Jorj\AppData\Local\Programs\Python\Python37\lib\site-packages\fitz\
↪pymupdf.py", line 2200, in __init__
    _pymupdf.Document_swiginit(self, _pymupdf.new_Document(filename, stream, filetype,␣
↪rect, width, height, fontsize))
RuntimeError: cannot open does-not-exist.pdf: No such file or directory
>>>
```

36.8 Coordinates

This is one of the most frequently used terms in this documentation. A **coordinate** generally means a pair of numbers `(x, y)` referring to some location, like a corner of a rectangle (*Rect*), a *Point* and so forth. The two values usually are floats, but there a objects like images which only allow them to be integers.

To actually *find* a coordinate's location, we also need to know the *reference* point for `x` and `y` - in other words, we must know where location `(0, 0)` is positioned. Once `(0, 0)` (the "origin") is known, we speak of a "coordinate system".

Several coordinate systems exist in document processing. For instance, the coordinate systems of a PDF page and the image created from it are **different**. We therefore need ways to *transform* coordinates from one system to another (and also back occasionally). This is the task of a *Matrix*. It is a mathematical function which works much like a factor that can be "multiplied" with a point or rectangle to give us the corresponding point / rectangle in another coordinate system. The inverse of a transformation matrix can be used to revert the transformation. Much like multiplying by some factor, say 3, can be reverted by dividing the result by 3 (or multiplying it with 1/3).

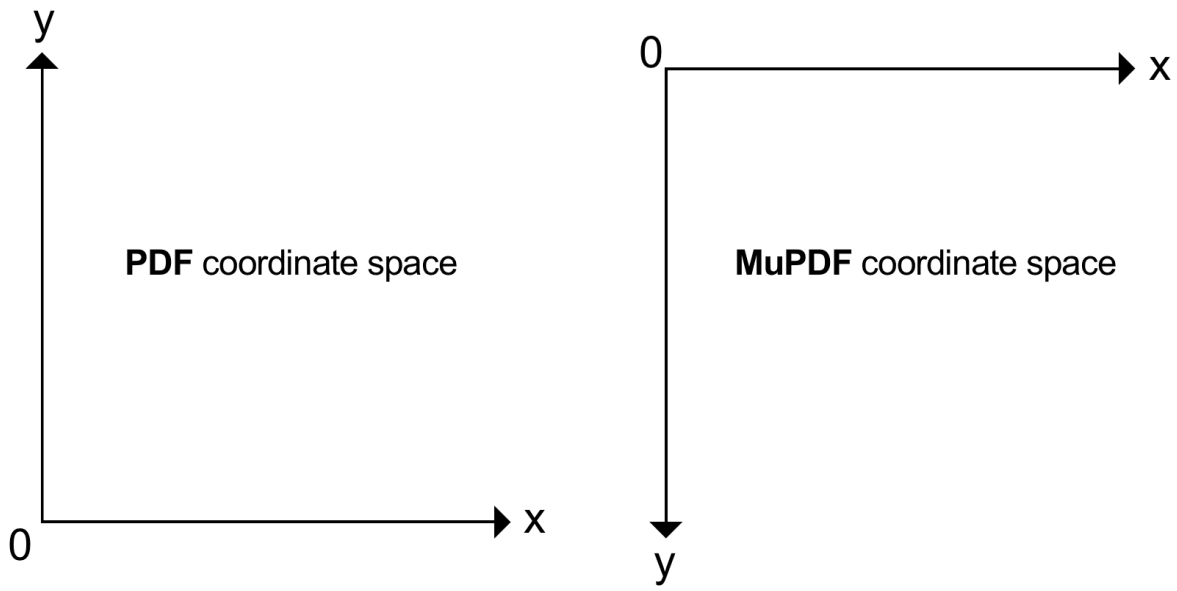
36.8.1 Coordinates and Images

Images have a coordinate system with integer coordinates. Origin `(0, 0)` is the top-left point. `x` values must be in `range(width)`, and `y` values in `range(height)`. Therefore, `y` values *increase* if we go *downwards*. For every image, there is only a **finite number** of coordinates, namely `width * height`. A location in an image is also called a "pixel".

- How **large** an image will be (in centimeters or inches) when e.g. printed, depends on additional information: the "resolution". This is measured in **DPI** (dots per inch, or pixels per inch). To find the printed size of some image, we therefore must divide its width and its height by the corresponding DPI values (there may separate ones for width and for height) and will get the respective number of inches.

36.8.2 Origin Point, Point Size and Y-Axis

In , the origin `(0, 0)` of a page is located at its **bottom-left point**. In , the origin `(0, 0)` of a page is located at its **top-left point**.



Coordinates are float numbers and measured in **points**, where:

- **one point equals 1/72 inches**.

Typical document page sizes are **ISO A4** and **Letter**. A **Letter** page has a size of **8.5 x 11 inches**, corresponding to **612 x 792 points**. In the coordinate system, the top-left point of a **Letter** page hence has the coordinate `(0, 792)` as **the y-axis points upwards**. Now we know our document size the coordinate system for the bottom right would be coordinate `(612, 792)` (and for this coordinate would then be `(612,0)`).

- Theoretically, there are **infinitely many** coordinate positions on a page. In practice however, at most the first 5 decimal places are sufficient for a reasonable precision.
- In , multiple document formats are supported - just being one among **over a dozen others**. Images are also supported as documents in (therefore having one page usually). This is one of the reasons why uses a coordinate system with the origin `(0, 0)` being the **top-left** point of any document page. **The y-axis points downwards**, like with images. Coordinates in in any case are floats, like in .
- A rectangle `Rect(0, 0, 100, 100)` for instance in (and thus) therefore is a square with edges of length 100 points (= 1.39 inches or 3.53 centimeters). Its top-left corner is the origin. To switch between the two coordinate systems to , every *Page* object has a *Page.transformation_matrix*. Its inverse can be used to compute a

rectangle's PDF coordinates. In this way we can conveniently find that `Rect(0, 0, 100, 100)` in is the same as `Rect(0, 692, 100, 792)` in . See this code snippet:

```
>>> page = doc.new_page(width=612, height=792)  # make new Letter page
>>> ptm = page.transformation_matrix
>>> # the inverse matrix of ptm is ~ptm
>>> pymupdf.Rect(0, 0, 100, 100) * ~ptm
Rect(0.0, 692.0, 100.0, 792.0)
```

36.9 CSS Support

For now, only a subset of CSS properties are supported.

The underlying C library MuPDF supports a subset of HTML4 and CSS2. The primary goal of the HTML/CSS support is to serve as a popular and convenient way to style text — not to faithfully reproduce websites in PDF.

36.9.1 What Works

The following list shows the supported properties, grouped by category.

Box Model & Layout

`margin`, `margin-top`, `margin-right`, `margin-bottom`, `margin-left`, `padding`, `padding-top`, `padding-right`, `padding-bottom`, `padding-left`, `width`, `height`, `display`, `position`, `top`, `right`, `bottom`, `left`, `inset`, `overflow-wrap`, `columns`

> **Note**
>
> The properties `position` & `display` are supported in a very limited way. Only the values `position: relative` and `display: block` are supported.

Border

`border`, `border-top`, `border-right`, `border-bottom`, `border-left`, `border-color`, `border-style`, `border-width`, `border-spacing`, `border-collapse`, `border-top-color`, `border-right-color`, `border-bottom-color`, `border-left-color`, `border-top-style`, `border-right-style`, `border-bottom-style`, `border-left-style`, `border-top-width`, `border-right-width`, `border-bottom-width`, `border-left-width`

Background

`background`, `background-color`

> **Note**
>
> Background images are not supported, but the `background` property can be used to set a background color for a text block, which is then rendered as a filled rectangle behind the text.

Font

`font`, `font-family`, `font-size`, `font-style`, `font-variant`, `font-weight`

Text

`color`, `letter-spacing`, `line-height`, `text-align`, `text-decoration`, `text-indent`, `text-transform`, `word-spacing`, `white-space`, `vertical-align`, `direction`, `hyphens`

List

`list-style`, `list-style-image`, `list-style-position`, `list-style-type`

Page

`page-break-before`, `page-break-after`, `orphans`, `widows`

Visibility

`visibility`

MuPDF-specific / WebKit extensions

`-mupdf-leading`, `-webkit-text-fill-color`, `-webkit-text-stroke-color`, `-webkit-text-stroke-width`

Other

`src` (for @font-face), `overflow-wrap`

36.9.2 What Doesn't Work

Modern CSS (CSS3+): no `flexbox`, `grid`, `custom properties` (–vars), `calc()`, `transitions`, `animations`, `float`, `clear`.

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTYSEVEN

APPENDIX 4: PERFORMANCE COMPARISON METHODOLOGY

This article documents the approach to measure *PyMuPDF's* performance and the tools and example files used to do comparisons.

The following three sections deal with different performance aspects:

- *Document Copying* - This includes opening and parsing *PDFs*, then writing them to an output file. Because the same basic activities are also used for joining (merging) *PDFs*, the results also apply to these use cases.
- *Text Extraction* - This extracts plain text from *PDFs* and writes it to an output text file.
- *Page Rendering* - This converts pages to image files looking identical to the pages. This ability is the basic prerequisite for using a tool in *Python GUI* scripts to scroll through documents. We have chosen a medium-quality (resolution 150 DPI) version.

Please note that in all cases the actual speed in dealing with structures is not directly measured: instead, the timings also include the durations of writing files to the operating system's file system. This cannot be avoided because tools other than do not offer the option to e.g., separate the image **creation** step from the following step, which **writes** the image into a file.

So all timings documented include a common, OS-oriented base effort. Therefore, performance **differences per tool are actually larger** than the numbers suggest.

37.1 Files used

A set of eight files is used for the performance testing. With each file we have the following information:

- **Name** of the file and download **link**.
- **Size** in bytes.
- Total number of **pages** in file.
- Total number of bookmarks (**Table of Contents** entries).
- Total number of **links**.
- **KB size** per page.
- **Textsize per page** is the amount text in the whole file in KB, divided by the number of pages.
- Any **notes** to generally describe the type of file.

Name	Size (bytes)	Page	TOC size	Links	KB/pag	Text-size/page	Notes
adobe.pdf	32,472,77	1,310	794	32,09	24	1,942	linearized, many links / bookmarks
artifex-website.pdf	31,570,73	47	46	2,035	656	3,538	graphics oriented
db-systems.pdf	29,326,35	1,241	0	0	23	2,142	
fontforge.pdf	8,222,384	214	31	242	38	1,058	mix of text & graphics
pandas.pdf	10,585,96	3,071	536	16,55	3	1,539	many pages
pymupdf.pdf	6,805,176	478	276	5,277	14	1,937	text oriented
pythonbook.pdf	9,983,856	669	198	1,953	15	1,929	
sample-50-MB-pdf-file.pdf	52,521,85	1	0	0	51,291	23,860	single page, graphics oriented, large file size

Note

adobe.pdf and **pymupdf.pdf** are clearly text oriented, **artifex-website.pdf** and **sample-50-MB-pdf-file.pdf** are graphics oriented. Other files are a mix of both.

37.2 Tools used

In each section, the same fixed set of files is being processed by a set of tools. The set of tools used per performance aspect however varies, depending on the supported tool features.

All tools are either platform independent, or at least can run on both, *Windows* and *Unix* / *Linux*.

Tool	Description
	The tool of this manual.
PDFrw	A pure *Python* tool, being used by *rst2pdf*, has interface to *ReportLab*.
PyPDF2	A pure *Python* tool with a large function set.
PDFMiner	A pure *Python* to extract text and other data from .
XPDF	A command line utility with multiple functions.
PikePDF	A *Python* package similar to *PDFrw*, but based on *C++* library *QPDF*.
PDF2JPG	A *Python* package specialized on rendering pages to *JPG* images.

37.3 Copying / Joining / Merging

How fast is a file read and its content parsed for further processing? The sheer parsing performance cannot directly be compared, because batch utilities always execute a requested task completely, in one go, front to end. *PDFrw* too, has a *lazy* strategy for parsing, meaning it only parses those parts of a document that are required in any moment.

To find an answer to the question, we therefore measure the time to copy a file to an output file with each tool, and do nothing else.

These are the *Python* commands for how each tool is used:

```
import pymupdf
doc = pymupdf.open("input.pdf")
doc.save("output.pdf")
```

PDFrw

```
doc = PdfReader("input.pdf")
writer = PdfWriter()
writer.trailer = doc
writer.write("output.pdf")
```

PikePDF

```
from pikepdf import Pdf
doc = Pdf.open("input.pdf")
doc.save("output.pdf")
```

PyPDF2

```
pdfmerge = PyPDF2.PdfMerger()
pdfmerge.append("input.pdf")
pdfmerge.write("output.pdf")
pdfmerge.close()
```

Observations

These are our run time findings in **seconds** along with a base rate summary compared to :

Name		PDFrw	PikePDF	PyPDF2
adobe.pdf	1.75	5.15	22.37	374.05
artifex-website.pdf	0.26	0.38	1.41	2.81
db-systems.pdf	0.15	0.8	1.68	2.46
fontforge.pdf	0.09	0.14	0.28	1.1
pandas.pdf	0.38	2.21	2.73	70.3
pymupdf.pdf	0.11	0.56	0.83	6.05
pythonbook.pdf	0.19	1.2	1.34	37.19
sample-50-MB-pdf-file.pdf	0.12	0.1	2.93	0.08
Total	**3.05**	**10.54**	**33.57**	**494.04**
Rate compared to PyMuPDF	1.0	3.5	11.0	162

37.4 Text Extraction

The following table shows plain text extraction durations. All tools have been used with their most basic functionality - i.e. no layout re-arrangements, etc.

Observations

These are our run time findings in **seconds** along with a base rate summary compared to :

Name		XPDF	PyPDF2	PDFMiner
adobe.pdf	2.01	6.19	22.2	49.15
artifex-website.pdf	0.18	0.3	1.1	4.06
db-systems.pdf	1.57	4.26	25.75	42.19
fontforge.pdf	0.24	0.47	2.69	4.2
pandas.pdf	2.41	10.54	25.38	76.56
pymupdf.pdf	0.49	2.34	6.44	13.55
pythonbook.pdf	0.84	2.88	9.28	24.27
sample-50-MB-pdf-file.pdf	0.27	0.44	8.8	13.29
Total	**8.01**	**27.42**	**101.64**	**227.27**
Rate compared to PyMuPDF	1.0	3.42	12.69	28.37

37.5 Page Rendering

We have tested rendering speed of against *pdf2jpg* and *XPDF* at a resolution of 150 DPI,

These are the *Python* commands for how each tool is used:

```
def ProcessFile(datei):
print "processing:", datei
doc=pymupdf.open(datei)
for p in pymupdf.Pages(doc):
    pix = p.get_pixmap(dpi=150)
    pix.save(f"t-{p.number}.png")
    pix = None
doc.close()
return
```

XPDF

```
pdftopng.exe -r 150 file.pdf ./
```

PDF2JPG

```
def ProcessFile(datei):
    print("processing:", datei)
    pdf2jpg.convert_pdf2jpg(datei, "images", pages="ALL", dpi=150)
    return
```

Observations

These are our run time findings in **seconds** along with a base rate summary compared to :

Name		XPDF	PDF2JPG
adobe.pdf	51.33	98.16	75.71
artifex-website.pdf	26.35	51.28	54.11
db-systems.pdf	84.59	143.16	405.22
fontforge.pdf	12.23	22.18	20.14
pandas.pdf	138.74	241.67	202.06
pymupdf.pdf	22.35	39.11	33.38
pythonbook.pdf	30.44	49.12	55.68
sample-50-MB-pdf-file.pdf	1.01	1.32	5.22
Total	**367.04**	**646**	**851.52**
Rate compared to PyMuPDF	1.0	1.76	2.32

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTYEIGHT

CHANGE LOG

Changes in version 1.28.2 (2026-08-06)

Fixed issues:

- **Fixed** 4670: scrub fails to remove hidden text after clean_contents stopped including line breaks (u2265 1.24.0)
- **Fixed** 4943: enh: applying redactions with image cropping for currently unsupported colorspaces
- **Fixed** 5030: find_tables() with layout enabled can return a zero-cell Table, and Table.bbox then raises "ValueError: min() iterable argument is empty"
- **Fixed** 5042: Page.get_texttrace() leaks None references u2014 Fatal Python error (none_dealloc) in long-running processes (1.27.2.3 & 1.28.0)
- **Fixed** 5044: Outline (TOC) parsing bug
- **Fixed** 5049: font subsetting segfaults in 1.28.0, regression from 1.27.2.2 #5049
- **Fixed** 5056: Option for reproducible/deterministic save output (omit MuPDF version banner + Producer)

Other:

- Use MuPDF-1.28.2.
- Output warning when legacy `fitz` module is imported.
- Cope better with markdown containing illegal utf8 sequences.
- Fixed building with PYMUPDF_SETUP_MUPDF_VS_UPGRADE.
- pymupdf.Page.find_tables(): * new args `use_layout: bool = True` `union: bool = False` `refine: bool = False`. * Improved speed.
- Retrospectively added fix for #4936 in release 1.28.0 below.
- Added windows-arm64 wheel.

Changes in version 1.28.0 (2026-06-29)

Fixed issues:

- **Fixed** 4114: ComboBox choice_values full of empty strings despite PDF having valid choices.
- **Fixed** 4950: remove_rotation() raises ValueError on widgets with empty/infinite rects
- **Fixed** 5001: Formulae incorrectly rendered as black boxes
- **Fixed** 5033: Annot.set_rotation(0) followed by Annot.update() throws AttributeError
- **Fixed** 4936: bug: incomplete redaction of vector graphics (line art)

Other:

- Use MuPDF-1.28.0.
- pymupdf.Document.__init__(): new arg *Archive* to support documents with archives.
- pymupdf.Document.convert_to_pdf(): also generate links.
- Fixed MacOS x64 platform tag - used to be macosx_10_9_x86_64, but we actually require MacOS 10.15, so platform tag is now macosx_10_15_x86_64.
- Support Windows builds with free thread python.
- pymupdf.Document.save() now saves non-PDF documents in PDF format.
- New method pymupdf.Document.apply_css().
- Pyodide wheel is available on pypi.org.

Changes in version 1.27.2.3 (2026-04-24)

- Fixed issues:
  - **Fixed** 4928: pymupdf.Document.scrub raises AttributeError for a document with annotations
  - **Fixed** 4942: bug: IndexError for Page.get_links after Page.clip_to_rect
  - **Fixed** 4954: get_drawings() returns incorrect lineJoin and width
  - **Fixed** 4958: bug: inserting rotated pages to another document messes up link coordinates
- Other:
  - Fixed incorrect generation of `lineJoin j` in PDF content, introduced in 1.27.2.2.
  - Allow build to (incorrectly) claim to be thread-safe, for #4760. See setup.py for details.
  - Use pypi.org's pipcl package instead of our own pipcl.py file.

Changes in version 1.27.2.2 (2026-03-20)

- Fixed issues:
  - **Fixed** 4902: Incorrect linewidth in elements returned by Page.get_texttrace()
  - **Fixed** 4932: "Page" has no attribute "find_tables" in PyMuPDF 1.27
- Other:
  - Added `Annot.__bool__()`.

Changes in version 1.27.2. (2026-03-10)

- Use MuPDF-1.27.2.
- Fixed issues:
  - **Fixed** 4903: Typing broken because of `*_forward_decl`
  - **Fixed** 3842: Partial OCR using "get_textpage_ocr" ignores image masks while extracting text
- Other:
  - Retrospectively marked #4907 as fixed in pymupdf-1.27.1.
  - Improved *`get_textpage_ocr()`*.

    For partial OCR, **all** page areas outside legible text are now OCRed, not just those within images. This means that OCR will now also be performed for vector graphics, and for text containing illegible characters.
  - Provide a Linux wheel for free-threading python, specifically cp314-cp314t-manylinux_2_28_x86_64.

Changes in version 1.27.1 (2026-02-11)

- Use MuPDF-1.27.1.
- Fixed issues:
  - **Fixed** 4599: page.cluster_drawings extract a lot of small clusters once upgraded to 1.26
  - **Fixed** 4751: Memory leaking in page.widgets()
  - **Fixed** 4762: Importing pymupdf make pillow segmentation fault for converting jp2 file on ArchLinux
  - **Fixed** 4790: Problem to delete pages on PDF
  - **Fixed** 4857: Package is missing py.typed file required for type checking
  - **Fixed** 4886: <IMG> width attribute behaviour seems wrong
  - **Fixed** 4907: signal 11:SIGSEGV while using display_list.get_textpage()
- Other:
  - Added `pymupdf.TEXT_CLIP`.
  - Removed support for mupdf < 1.26.
  - New arg `raise_on_repair` in *Document.save()*.
  - New method *Document.repair()*.

Changes in version 1.26.7 (2025-12-11)

- Use MuPDF-1.26.12.
  - **Fixed** 4801: Build failure dumping all environment variables

Other:

- Retrospectively mark 4756 as fixed in 1.26.6.
- Improved safety of `pymupdf embed-extract`. This now refuses to write to an existing file or outside current directory, unless `-output` or new flag `-unsafe` is specified.

Changes in version 1.26.6 (2025-11-05)

- Use MuPDF-1.26.11.
- Supported Python versions are now 3.10-3.14.
- Fixed issues:
  - **Fixed** 4699: cannot find ExtGState resource
  - **Fixed** 4712: Crash with "corrupted double-linked list"
  - **Fixed** 4720: Memory leaking in rewrite_images?
  - **Fixed** 4742: 'Rect' object has no attribute 'get_area'
  - **Fixed** 4746: Document.__init__() got an unexpected keyword argument 'encoding'
  - **Fixed** 4756: swig –version doesn't work in all versions of swig; -version should be used instead

Changes in version 1.26.5 (2025-10-10)

- Use MuPDF-1.26.10.
- Fixed issues:
  - **Fixed** 2883: Improve the Python type annotations for fitz_new
  - **Fixed** 4507: Bugs in pyodide

  - **Fixed** 4613: Thai and number blocks are not auto-scaled and get wrong hyphen when using in insert_htmlbox
  - **Fixed** 4700: pymupdf.open() processes .zip file without raising
  - **Fixed** 4716: Problems with unreadable characters
- Other:
  - Supported Python versions are now 3.9-3.14.
  - We now define all class methods explicitly instead of with dynamic assignment; this improves type hints.
  - Removed `pymupdf.utils.Shape` class, was duplicate of `pymupdf.Shape`.
  - Allow use of cibuildwheel to build and test on Pyodide.
  - Fixed various Pyodide bugs.
  - In documentation, added section about Linux wheels and glibc compatibility.
  - Improved documentation of pymupdf.open()'s <filetype> arg.
  - Retrospectively mark 4544 as fixed in 1.26.4.

Changes in version 1.26.4 (2025-08-25)

- Use MuPDF-1.26.7.
- Fixed issues:
  - **Fixed** 3806: pdf to image rendering ignore optional content offs
  - **Fixed** 4388: Incorrect PixMap from page due to cached data from other PDF
  - **Fixed** 4457: Wrong characters displayed after font subsetting (w/ native method)
  - **Fixed** 4462: delete_pages() does not accept a single int
  - **Fixed** 4533: Open PDF error segmentation fault
  - **Fixed** 4544: About pdf_clip_page
  - **Fixed** 4565: MacOS uses Tesseract and not Tesseract-OCR
  - **Fixed** 4571: Broken merged pdfs.
  - **Fixed** 4590: TypeError in utils.py scrub(): annot.update_file(buffer=...) is invalid
  - **Fixed** 4614: Intercept bad widgets when inserting to another PDF
  - **Fixed** 4639: pymupdf.mupdf.FzErrorGeneric: code=1: Director error: <class 'AttributeError'>: 'JM_new_bbox_device_Device' object has no attribute 'layer_name'
- Other:
  - Check that #4392 `Segfault when running with pytest and -Werror` is fixed if PyMuPDF is built with swig>=4.4.
  - Add *`Page.clip_to_rect()`*.
  - Improved search for Tesseract data.
  - Retrospectively mark #4496 as fixed in 1.26.1.
  - Retrospectively mark #4503 as fixed in 1.26.3.
  - Added experimental support for Graal.

Changes in version 1.26.3 (2025-07-02)

- Use MuPDF-1.26.3.
- Fixed issues:
  - **Fixed** 4462: delete_pages() does not accept a single int
  - **Fixed** 4503: Undetected character styles
  - **Fixed** 4527: Rect.intersects() is much slower than necessary
  - **Fixed** 4564: Possible encoding issue in PDF metadata
  - **Fixed** 4575: Bug with IRect contains method
- Other:
  - Class Shape is now available as pymupdf.Shape.
  - Added table cell markdown support.

Changes in version 1.26.2

[Skipped.]

Changes in version 1.26.1 (2025-06-11)

- Use MuPDF-1.26.2.
- Fixed issues:
  - **Fixed** 4520: show_pdf_page does not like empty pages created by new_page
  - **Fixed** 4524: fitz.get_text ignores 'pages' kwarg
  - **Fixed** 4412: Regression? Spurious error? in insert_pdf in v1.25.4
  - **Fixed** 4496: pymupdf4llm with pymupdfpro
- Other:
  - Partial fix for 4503: Undetected character styles
  - New method *Document.rewrite_images()*, useful for reducing file size, changing image formats, or converting color spaces.
  - *Page.get_text()*: restrict positional args to match docs.
  - Removed bogus definition of class *Shape*.
  - Removed release date from module, docs and changelog. * `pymupdf.pymupdf_date` and `pymupdf.VersionDate` are now both None. * They will be removed in a future release.

Changes in version 1.26.0 (2025-05-22)

- Use MuPDF-1.26.1.
- Fixed issues:
  - **Fixed** 4324: cluster_drawings() fails to cluster horizontal and vertical thin lines
  - **Fixed** 4363: Trouble with searching
  - **Fixed** 4404: IndexError in page.get_links()
  - **Fixed** 4412: Regression? Spurious error? in insert_pdf in v1.25.4
  - **Fixed** 4423: pymupdf.mupdf.FzErrorFormat: code=7: cannot find object in xref error encountered after version 1.25.3
  - **Fixed** 4435: get_pixmap method stuck on one page

  - **Fixed** 4439: New Xml class from data does not work - bug in code
  - **Fixed** 4445: Broken XREF table incorrectly repaired
  - **Fixed** 4447: Stroke color of annotations cannot be correctly set
  - **Fixed** 4479: set_layer_ui_config() toggles all layers rather than just one
  - **Fixed** 4505: Follow Widget flag values up its parent structure
- Other:
  - Partial fixed for 4457: Wrong characters displayed after font subsetting (w/ native method)
  - Support image stamp annotations.
  - Support recoloring pages.
  - Added example of using Django's file storage API to open files with pymupdf.
  - Clarified FreeText annotation color options. We now raise an exception if an attempt is made to set attributes that can not be supported.
  - Fixed potential segv in Pixmap.is_unicolor().
  - Added runtime assert that that PyMuPDF and MuPDF were built with compatible NDEBUG settings (related to 4390).
  - Simplified handling of filename/filetype when opening documents.
  - Removed PDF linearization support. * Calls to *Document.save()* with `linear` set to true will now raise an exception. * See https://artifex.com/blog/mupdf-removes-linearisation for more information.

Changes in version 1.25.5 (2025-03-31)

- Fixed issues:
  - **Fixed** 4372: Text insertion fails due to missing /Resources object
  - **Fixed** 4400: Infinite loop in fill_textbox
  - **Fixed** 4403: Unable to get_text() - layer/clip nesting too deep
  - **Fixed** 4415: PDF page is mirrored, origin is at bottom-left
- Other:
  - Use MuPDF-1.25.6.
  - Fixed MuPDF SEGV on MacOS with particular fonts.
  - Fixed *Annot.get_textpage()*'s `clip` arg.
  - Fixed Python-3.14 (pre-release) build error.

Changes in version 1.25.4 (2025-03-14)

- Use MuPDF-1.25.5.
- Fixed issues:
  - **Fixed** 4079: Unexpected result for apply_redactions()
  - **Fixed** 4224: MuPDF error: format error: negative code in 1d faxd
  - **Fixed** 4303: page.get_image_info() returns outdated cached results after replacing image
  - **Fixed** 4309: FzErrorFormat Error When Deleting First Page

  - **Fixed** 4336: Major Performance Regression: pix.color_count is 150x slower in version 1.25.3 compared to 1.23.8
  - **Fixed** 4341: Invalid label retrieval when /Kids is an array of multiple /Nums
- Other:
  - Fixed handling of duplicate widget names when joining PDFs (PR #4347).
  - Improved Pyodide build.
  - Avoid SWIG-related build errors with Python-3.13 by disabling PY_LIMITED_API.

Changes in version 1.25.3 (2025-02-06)

- Use MuPDF-1.25.4.
- Fixed issues:
  - **Fixed** 4139: Text color numbers change between 1.24.14 and 1.25.0
  - **Fixed** 4141: Some insertion methods fails for pages without a /Resources object
  - **Fixed** 4180: Search problems
  - **Fixed** 4182: Text coordinate extraction error
  - **Fixed** 4245: Highlighting issue distorted on recent versions
  - **Fixed** 4254: add_freetext_annot is drawing text outside the annotation box
- Other:
  - In annotations: * Added support for subtype FreeTextCallout. * Added support for rich text.
  - Added miter_limit arg to insert_text*() to allow suppression of spikes caused by long miters.
  - Add Widget Support to *Document.insert_pdf()*.
  - Add `bibi` to span dicts.
  - Add `synthetic' to char dict.
  - Fixed Pyodide builds.

Changes in version 1.25.2 (2025-01-17)

- Fixed issues:
  - **Fixed** 4055: "Yes" for all checkboxes does not work for all PDF rendering engines.
  - **Fixed** 4155: samples_mv is unsafe
  - **Fixed** 4162: Got AttributeError, when tried to add Signature field
  - **Fixed** 4186: Incorrect handling of JPEG with color space CMYK image extraction
  - **Fixed** 4195: Pixmaps that are inverted and have an alpha channel are not rendered properly
  - **Fixed** 4225: pixmap.pil_save() fails due to colorspace definition
  - **Fixed** 4232: Incorrect Font style and Size
- Other:
  - Use Python's built-in glyphname <> unicode conversion.
  - Improve speed of pixmap color inversion.
  - Add new `char_flags` member to span dictionary, for example allows detection of invisible text.

  - Detect image masks in TextPage output.
  - Added *Pixmap.pil_image()*.

Changes in version 1.25.1 (2024-12-11)

- Use MuPDF-1.25.2.
- Fixed issues:
  - **Fixed** 4125: memory leak while convert Pixmap's colorspace
  - **Fixed** 4034: Possible regression in pdf cleaning during save.

Changes in version 1.25.0 (2024-12-05)

- Use MuPDF-1.25.1.
- Fixed issues:
  - **Fixed** 4026: page.get_text('blocks') output two piece of very similar text with different bbox
  - **Fixed** 4004: Segmentation Fault When Updating PDF Form Field Value
  - **Fixed** 3887: Subset Fonts problem using Fallback Font
  - **Fixed** 3886: Another issue with destroying PDF when inserting html
  - **Fixed** 3751: apply_redactions causes part of the page content to be hidden / transparent

Changes in version 1.24.14 (2024-11-19)

- Use MuPDF-1.24.11.
- Fixed issues:
  - **Fixed** 3448: get_pixmap function removes the table and leaves just the content behind
  - **Fixed** 3758: Got "malloc(): unaligned tcache chunk detected Aborted (core dumped)" while using add_redact_annot/apply_redactions
  - **Fixed** 3813: Stories: Ordered list count broken with nested unordered list
  - **Fixed** 3933: font.valid_codepoints() - malfunction
  - **Fixed** 4018: PyMuPDF hangs when iterating over zero page PDF pages backwards
  - **Fixed** 4043: fullcopypage bug
  - **Fixed** 4047: Segmentation Fault in add_redact_annot
  - **Fixed** 4050: Content of dict returned by doc.embfile_info() does not fit to documentation
- Other:
  - Ensure that words from *Page.get_text()* never contain RTL/LTR char mixtures.
  - Fix building with system MuPDF.
  - Add dot product for points and vectors.

Changes in version 1.24.13 (2024-10-29)

- Fixed issues:
  - **Fixed** 3848: Piximap program crash
  - **Fixed** 3950: Unable to consistently extract field labels from PDFs
  - **Fixed** 3981: PyMuPDF 1.24.12 with pyinstaller throws error.

  - **Fixed** 3994: pix.color_topusage raise Segmentation fault (core dumped)

Changes in version 1.24.12 (2024-10-21)

- Fixed issues:
  - **Fixed** 3914: Ability to print MuPDF errors to logging instead of stdout
  - **Fixed** 3916: insert_htmlbox error: int too large to convert to float
  - **Fixed** 3950: Unable to consistently extract field labels from PDFs
- Supported Python versions are now 3.9-3.13.
  - Dropped support for Python-3.8 because end-of-life.
  - Added support for Python-3.13 because now released.
  - See: https://devguide.python.org/versions/

Changes in version 1.24.11 (2024-10-03)

- Use MuPDF-1.24.10.
- Fixed issues:
  - **Fixed** 3624: Pdf file transform to image have a black block
  - **Fixed** 3859: doc.need_appearances() fails with "AttributeError: module 'pymupdf.mupdf' has no attribute 'PDF_TRUE' "
  - **Fixed** 3863: apply_redactions() does not work as expected
  - **Fixed** 3905: open stream can raise a FzErrorFormat error instead of FileDataError
- Wheels now use the Python Stable ABI:
  - There is one PyMuPDF wheel for each platform.
  - Each wheel works with all supported Python versions.
  - Each wheel is built using the oldest supported Python version (currently 3.8).
  - There is no PyMuPDFb wheel.
- Other:
  - Improvements to get_text_words() with sort=True.
  - Tests now always get the latest versions of required Python packages.
  - Removed dependency on setuptools.
  - Added item to PyMuPDF-1.24.10 changes below - fix of #3630.

Changes in version 1.24.10 (2024-09-02)

- Use MuPDF-1.24.9.
- Fixed issues:
  - **Fixed** 3450: get_pixmap function takes too long to process
  - **Fixed** 3569: Invalid OCGs not ignored by SVG image creation
  - **Fixed** 3603: ObjStm compression and PDF linearization doesn't work together
  - **Fixed** 3650: Linebreak inserted between each letter
  - **Fixed** 3661: Update Document to check the /XYZ len

  - **Fixed** 3698: documentation issue - old code in the annotations documentation
  - **Fixed** 3705: Document.select() behaves weirdly in some particular kind of pdf files
  - **Fixed** 3706: extend Document.__getitem__ type annotation to reflect that the method also accepts slices
  - **Fixed** 3727: Method get_pixmap() make the program exit without any exceptions or messages
  - **Fixed** 3767: Cannot get Tessdata with Tesseract-OCR 5
  - **Fixed** 3773: Link.set_border gives TypeError: '<' not supported between instances of 'NoneType' and 'int'
  - **Fixed** 3774: fitz.__version__` does not work anymore
  - **Fixed** 3789: ValueError: not enough values to unpack (expected 3, got 2) is thrown when call insert_pdf
  - **Fixed** 3820: class improves namedDest handling
  - **Fixed** 3630: page.apply_redactions gives unwanted black rectangle
- Other:
  - Object streams and linearization cannot be used together; attempting to do so will raise an exception. (#3603)
  - Fixed handling of non-existing /Contents object.

Changes in version 1.24.9 (2024-07-24)

- Use MuPDF-1.24.8.

Changes in version 1.24.8 (2024-07-22)

- Fixed issues:
  - **Fixed** 3636: API documentation for the open function is not obvious to find.
  - **Fixed** 3654: docx parsing was broken in 1.24.7
  - **Fixed** 3677: Unable to extract subset font name using the newer versions of PyMuPDF : 1.24.6 and 1.24.7.
  - **Fixed** 3687: Page.get_text results in AssertionError for epub files

Other:

- Fixed various spelling mistakes spotted by codespell.
- Improved how we modify MuPDF's default configuration on Windows.
- Make text search to work with ligatures.

Changes in version 1.24.7 (2024-06-26)

- Fixed issues:
  - **Fixed** 3615: Document.pagemode or Document.pagelayout crashes for epub files
  - **Fixed** 3616: not last version reported

Changes in version 1.24.6 (2024-06-25)

- Use MuPDF-1.24.4
- Fixed issues:
  - **Fixed** 3599: Story.fit_width() has a weird line
  - **Fixed** 3594: Garbled extraction for Amazon Sustainability Report
  - **Fixed** 3591: 'width' in Page.get_drawings() returns width equal as 0

  - **Fixed** 3561: ZeroDivisionError: float division by zero with page.apply_redactions()
  - **Fixed** 3559: SegFault 11 when empty H1 H2 H3 H4 etc element is used in insert_htmlbox
  - **Fixed** 3539: Add dotted gridline detection to table recognition
  - **Fixed** 3519: get_toc(simple=False) AttributeError: 'Outline' object has no attribute 'rect'
  - **Fixed** 3510: page.get_label() gets wrong label on the first page of doc
  - **Fixed** 3494: 1.24.2/1.24.3: spurious characters introduced when using subset_fonts and insert_pdf
  - **Fixed** 3470: subset_fonts error exit without exception/warning
  - **Fixed** 3400: set_toc alters link coordinates for some rotated pages on pymupdf 1.24.2
  - **Fixed** 3347: Incorrect links to points on pages having different heights
  - **Fixed** 3237: Set_metadata() does not work
  - **Fixed** 3493: Isolate PyMuPDF from other libraries; issues when PyMuPDF is loaded with other libraries like GdkPixbuf
- Other:
  - Fixed concurrent use of PyMuPDF caused by use of constant temporary filenames.
  - Add musllinux x86_64 wheels to release.
  - Added clearer version information:
    * `pymupdf.pymupdf_version`.
    * `pymupdf.mupdf_version`.
    * `pymupdf.pymupdf_date`.

Changes in version 1.24.5 (2024-05-30)

- Fixed issues:
  - **Fixed** 3479: regression: fill_textbox: IndexError: pop from empty list
  - **Fixed** 3488: set_toc method error
- Other:
  - Some more fixes to use MuPDF floating formatting.
  - Removed/disabled some unnecessary diagnostics.
  - Fixed utils.do_links() crash.
  - Experimental new functions `pymupdf.apply_pages()` and `pymupdf.get_text()`.
  - Addresses wrong label generation for label styles "a" and "A".

Changes in version 1.24.4 (2024-05-16)

- **Fixed** 3418: Re-introduced bug, text align add_redact_annot
- **Fixed** 3472: insert_pdf gives SystemError
- Other:
  - Fixed sysinstall test failing to remove all of prior installation before new install.
  - Fixed `utils.do_links()` crash.
  - Correct *TextPage* creation Code.

  - Unified various diagnostics.
  - Fix bug in `page_merge()`.

Changes in version 1.24.3 (2024-05-09)

- The Python module is now called `pymupdf`. `fitz` is still supported for backwards compatibility.
- Use MuPDF-1.24.2.
- Fixed issues:
  - **Fixed** 3357: PyMuPDF==1.24.0 will hanging when using page.get_text("text")
  - **Fixed** 3376: Redacting results are not as expected in 1.24.x.
  - **Fixed** 3379: Documentation mismatch for get_text_blocks return value order.
  - **Fixed** 3381: Contents stream contains floats in scientific notation
  - **Fixed** 3402: Cannot add Widgets containing inter-field-calculation JavaScript
  - **Fixed** 3414: missing attribute set_dpi()
  - **Fixed** 3430: page.get_text() cause process freeze with certain pdf on v1.24.2
- Other:
  - New/modified methods:
    - *Page.remove_rotation()*: new, set page rotation to zero while keeping appearance.
  - Fixed some problems when checking for PDF properties.
  - Fixed pip builds from sdist (see discussion 3360: Alpine linux docker build failing "No matching distribution found for pymupdfb==1.24.1").

Changes in version 1.24.2 (2024-04-17)

- Removed obsolete classic implementation from releases (previously available as module `fitz_old`).
- Fixed issues:
  - **Fixed** 3331: Document.pages() is incorrectly type-hinted
  - **Fixed** 3354: PyMuPDF==1.24.1: AttributeError: property 'metadata' of 'Document' object has no setter
- Other:
  - New/modified methods:
    - *Document.bake()*: new, make annotations / fields permanent content.
    - *Page.cluster_drawings()*: new, identifies drawing items (i.e. vector graphics or line-art) that belong together based on their geometrical vicinity.
    - *Page.apply_redactions()*: added new parameter *text*.
    - *Document.subset_fonts()*: use MuPDF's `pdf_subset_fonts()` instead of PyMuPDF code.
  - The *Document* class now supports page numbers specified as slices.
  - Avoid causing MuPDF warnings.

Changes in version 1.24.1 (2024-04-02)

- Fixed issues:
  - **Fixed** 3278: apply_redactions moves some unredacted text
  - **Fixed** 3301: Be more permissive when classifying links as kind LINK_URI

  - **Fixed** 3306: Text containing capital 'ET' not appearing as annotation

- Other:
  - Use MuPDF-1.24.1.
  - Support ObjStm Compression. Methods *Document.save()*, *Document.ez_save()* and `Document.write()` now support new parameters `use_objstm`, compression_effort` and `preserve_metadata`.

Changes in version 1.24.0 (2024-03-21)

- Fixed issues:
  - **Fixed** 3281: Preparing metadata (pyproject.toml) did not run successfully
  - **Fixed** 3279: PyMuPDF no longer builds in Alpine Linux
  - **Fixed** 3257: apply_redactions() deleting text outside of annotated box
  - **Fixed** 3216: AttributeError: 'Annot' object has no attribute '__del__'
  - **Fixed** 3207: get_drawings's items is missing line from h path operator
  - **Fixed** 3201: Memory leaks when merging PDFs
  - **Fixed** 3197: page.get_text() returns hexadecimal text for some characters
  - **Fixed** 3196: Remove text not working in 1.23.25 version vs 1.20.2
  - **Fixed** 3172: PDF's 45º lines disappearing in png conversion
  - **Fixed** 3135: Do not log warnings to stdout
  - **Fixed** 3125: get_pixmap method stuck on one page and runs forever
  - **Fixed** 2964: There is an issue with the image generated by the page.get_pixmap() function
- Other:
  - Use MuPDF-1.24.0.
  - Add support for redacting vector graphics.
  - Several fixes for table module
    * Add new method for outputting the table as a markdown string.
    * Address errors in computing the table header object:

      We now allow None as the cell value, because this will be resolved where needed (e.g. in the pandas DataFrame).

      We previously tried to enforce rect-like tuples in all header cell bboxes, however this fails for tables with all-None columns. This fix enables this and constructs an empty string in the corresponding cell string.

      We now correctly include start / stop points of lines in the bbox of the clustered graphic. We previously joined the line's rectangle - which had no effect because this is always empty.
  - Improved exception text if we fail to open document.
  - Fixed build with new libclang 18.

Changes in version 1.23.26 (2024-02-29)

- Fixed issues:
  - **Fixed** 3199: Add entry_points to setuptools configuration to provide command-line console scripts
  - **Fixed** 3209: Empty vertices in ink annotation

- Other:
  - Improvements to table detection:
    - Improved check for empty tables, fixes bugs when determining table headers.
    - Improved computation of enveloping vector graphic rectangles.
    - Ignore more meaningless "pseudo" tables
  - Install command-line 'pymupdf' command that runs fitz/__main__.py.
  - Don't overwrite MuPDF's config.h when building on non-Windows.
  - Fix *Story* constructor's *Archive* arg to match docs - now accepts a single *Archive* constructor arg.
  - Do not include MuPDF source in sdist; will be downloaded automatically when building.

Changes in version 1.23.25 (2024-02-20)

- Fixed issues:
  - **Fixed** 3182: Pixmap.invert_irect argument type error
  - **Fixed** 3186: extractText() extracts broken text from pdf
  - **Fixed** 3191: Error on .find_tables()
- Other:
  - When building, be able to specify python-config directly, with environment variable `PIPCL_PYTHON_CONFIG`.

Changes in version 1.23.24 (2024-02-19)

- Fixed issues:
  - **Fixed** 3148: Table extraction - vertical text not handled correctly
  - **Fixed** 3179: Table Detection: Incorrect Separation of Vector Graphics Clusters
  - **Fixed** 3180: Cannot show optional content group: AttributeError: module 'fitz.mupdf' has no attribute 'pdf_array_push_drop'
- Other:
  - Be able to test system install using `sudo pip install` instead of a venv.

Changes in version 1.23.23 (2024-02-18)

- Fixed issues:
  - **Fixed** 3126: Initialising Archive with a pathlib.Path fails.
  - **Fixed** 3131: Calling the next attribute of an Annot raises a "No attribute .parent" warning
  - **Fixed** 3134: Using an IRect as clip parameter in Page.get_pixmap no longer works since 1.23.9
  - **Fixed** 3140: PDF document stays in use after closing
  - **Fixed** 3150: doc.select() hangs on this doc.
  - **Fixed** 3163: AssertionError on using fitz.IRect
  - **Fixed** 3177: fitz.Pixmap(None, pix) Unrecognised args for constructing Pixmap
- Other:

  - Improved `Document.select()` by using new MuPDF function `pdf_rearrange_pages()`. This is a more complete (and faster) implementation of what needs to be done here in that not only pages will be rearranged, but also consequential changes will be made to the table of contents, links to removed pages and affected entries in the Optional Content definitions.
  - *TextWriter.appendv()*: added `small_caps` arg.
  - Fixed some valgrind errors with MuPDF master.
  - Fixed `Document.insert_image()` when build with MuPDF master.

Changes in version 1.23.22 (2024-02-12)

- Fixed issues:
  - **Fixed** 3143: Difference in decoding of OCGs names between doc.get_ocgs() and page.get_drawings()
  - **Fixed** 3139: Pixmap resizing needs positional arg "clip" - even if None.
- Other:
  - Removed the use of MuPDF function `fz_image_size()` from PyMuPDF.

Changes in version 1.23.21 (2024-02-01)

- Fixed issues:
- Other:
  - Fixed bug in set_xml_metadata(), PR 3112: Fix pdf_add_stream metadata error
  - Fixed lack of `.parent` member in *TextPage* from *Annot.get_textpage()*.
  - Fixed bug in *Page.add_widget()*.

Changes in version 1.23.20 (2024-01-29)

- Bug fixes:
  - **Fixed** 3100: Wrong internal property accessed in get_xml_metadata
- Other:
  - Significantly improved speed of *Document.get_toc()*.

Changes in version 1.23.19 (2024-01-25)

- Bug fixes:
  - **Fixed** 3087: Exception in insert_image with mask specified
  - **Fixed** 3094: TypeError: '<' not supported between instances of 'FzLocation' and 'int' in doc.delete_pages
- Other:
  - When finding tables:
    - Allow addition of user-defined "virtual" vector graphics when finding tables.
    - Confirm that the enveloping bboxes of vector graphics are inside the clip rectangle.
    - Avoid slow finding of rectangle intersections.
  - Added *Font.bbox* property.

Changes in version 1.23.18 (2024-01-23)

- Bug fixes:
  - **Fixed** 3081: doc.close() not closing the document

- Other:
  - Reduced size of sdist to fit on pypi.org (by reducing size of two test files).
  - Fix *Annot.file_info()* if no `Desc` item.

Changes in version 1.23.17 (2024-01-22)

- Bug fixes:
  - **Fixed** 3062: page_rotation_reset does not return page to original rotation
  - **Fixed** 3070: update_link(): AttributeError: 'Page' object has no attribute 'super'
- Other:
  - Fixed bug in *Page.links()* (PR #3075).
  - Fixed bug in *Page.get_bboxlog()* with layers.
  - Add support for timeouts in scripts/ and tests/run_compound.py.

Changes in version 1.23.16 (2024-01-18)

- Bug fixes:
  - **Fixed** 3058: Pixmap created from CMYK JPEG delivers RGB format
- Other:
  - In table detection strategy "lines_strict", exclude fill-only vector graphics.
  - Fixed sysinstall test failure.
  - In documentation, update feature matrix with item about text writing.

Changes in version 1.23.15 (2024-01-16)

- Bug fixes:
  - **Fixed** 3050: python3.9 pix.set_pixel has something wrong in c.append(ord(i))
- Other:
  - Improved docs for Page.find_tables().

Changes in version 1.23.14 (2024-01-15)

- Bug fixes:
  - **Fixed** 3038: JM_pixmap_from_display_list > Assertion Error : Checking for wrong type
  - **Fixed** 3039: Issue with doc.close() not closing the document in PyMuPDF
- Other:
  - Ensure valid "re" rectangles in *Page.get_drawings()* with derotated pages.

Changes in version 1.23.13 (2024-01-15)

- Bug fixes:
  - **Fixed** 2979: list index out of range in to_pandas()
  - **Fixed** 3001: Calling find_tables() on one document alters the bounding boxes of a subsequent document
- Other:
  - Fixed *Rect.height* and *Rect.width* to never return negative values.
  - Fixed `TextPage.extractIMGINFO()`'s returned `dictkey_yres` value.

Changes in version 1.23.12 (2024-01-12)

- – **Fixed** 3027: Page.get_text throws Attribute Error for 'parent'

Changes in version 1.23.11 (2024-01-12)

- Fixed some Pixmap construction bugs.
- Fixed Pixmap.yres().

Changes in version 1.23.10 (2024-01-12)

- Bug fixes:
  - **Fixed** 3020: Can't resize a PixMap
- Other:
  - Fixed Page.delete_image().

Changes in version 1.23.9 (2024-01-11)

- Default to new "rebased" implementation.
  - The old "classic" implementation is available with `import fitz_old as fitz`.
  - For more information about why we are changing to the rebased implementation, see: https://github.com/pymupdf/PyMuPDF/discussions/2680
- Use MuPDF-1.23.9.
- Bug fixes (rebased implementation only):
  - **Fixed** 2911: Page.derotation_matrix returns a tuple instead of a Matrix with rebased implementation
  - **Fixed** 2919: Rebased version: KeyError in resolve_names when merging pdfs
  - **Fixed** 2922: New feature that allows inserting named-destination links doesn't work
  - **Fixed** 2943: ZeroDivisionError: float division by zero when use apply_redactions()
  - **Fixed** 2950: Shelling out to pip during tests is problematic
  - **Fixed** 2954: Replacement unicode character in text extraction
  - **Fixed** 2957: apply_redactions() moving text
  - **Fixed** 2961: Passing a string as a page number raises IndexError instead of TypeError.
  - **Fixed** 2969: annot.next throws AttributeError
  - **Fixed** 2978: 1.23.9rc1: module 'fitz.mupdf' has no attribute 'fz_copy_pixmap_rect'
  - **Fixed** 2907: segfault trying to call clean_contents on certain pdfs with python 3.12
  - **Fixed** 2905: SystemError: <built-in function TextPage_extractIMGINFO> returned a result with an exception set
  - **Fixed** 2742: Segmentation Fault when inserting three (but not two) copies of the same source page into one destination page
- Other:
  - Add optional setting of opacity to *Page.insert_htmlbox()*.
  - Fixed issue with add_redact_annot() mentioned in #2934.
  - Fixed *Page.rotation()* to return 0 for non-PDF documents instead of raising an exception.
  - Fixed internal quad detection to cope with any Python sequence.

  - Fixed rebased `fitz.pymupdf_version_tuple` - was previously set to mupdf version.
  - Improved support for Linux system installs, including adding regular testing on Github.
  - Add missing `flake8` to `scripts/gh_release.py:test_packages`.
  - Use newly public functions in MuPDF-1.23.8.
  - Improved `scripts/test.py` to help investigation of MuPDF issues.

Changes in version 1.23.8 (2023-12-19)

- Bug fixes (rebased implementation only):
  - **Fixed** 2634: get_toc and set_toc do not behave consistently for rotated pages
  - **Fixed** 2861: AttributeError in getLinkDict during PDF Merge
  - **Fixed** 2871: KeyError in getLinkDict during PDF merge
  - **Fixed** 2886: Error in Skeleton for Named Link Destinations
- Bug fixes (rebased and classic implementations):
  - **Fixed** 2885: pymupdf find tables too slow
- Other:
  - Rebased implementation:
    - *Page.insert_htmlbox()*: new, much more powerful alternative to *Page.insert_textbox()* or *TextWriter.fill_textbox()*, using *Story*.
    - `Story.fit*()`: new methods for fitting a Story into an expanded rect.
    - *Story.write_with_links()*: add support for external links.
    - `Document.language()`: fixed to use MuPDF's new `mupdf.fz_string_from_text_language2()`.
    - *Document.subset_fonts()* - fixed.
    - Fixed internal `Archive._add_treeitem()` method.
    - Fixed `fitz_new.__doc__` to contain PyMuPDF and Python version information, and OS name.
    - Removed use of `(*args, **kwargs)` in API, we now specify keyword args explicitly.
    - Work with new MuPDF Python exception classes.
  - Fixed bug where *button_states()* returns None when `/AP` points to an indirect object.
  - Fixed pillow test to not ignore all errors, and install pillow when testing.
  - Added test for `fitz.css_for_pymupdf_font()` (uses package `pymupdf-fonts`).
  - Simplified Github Actions test specifications.
  - Updated `tests/README.md`.

Changes in version 1.23.7 (2023-11-30)

- Bug fixes in rebased implementation, not fixed in classic implementation:
  - **Fixed** 2232: Geometry helper classes should support keyword arguments
  - **Fixed** 2788: Problem with get_toc in pymupdf 1.23.6
  - **Fixed** 2791: Experiencing small memory leak in save()
- Bug fixes (rebased and classic implementations):

  - **Fixed** 2736: Failure when set cropbox with mediabox negative value
  - **Fixed** 2749: RuntimeError: cycle in structure tree
  - **Fixed** 2753: Story.write_with_links will ignore everything after the first "page break" in the HTML.
  - **Fixed** 2812: find_tables on landscape page generates reversed text
  - **Fixed** 2829: [cannot create /Annot for kind] is still printed despite #2345 is closed.
  - **Fixed** 2841: Unexpected KeyError when using scrub with fitz_new
- Use MuPDF-1.23.7.
- Other:
  - Rebased implementation:
    * Added flake8 code checking to test suite, and made various fixes.
    * Disable diagnostics during Document constructor to match classic implementation.
  - Additional fix to 2553: Invalid characters in versions >= 1.22
  - Fixed MuPDF Bug 707324: Story: HTML table row background color repeated incorrectly
  - Added `scripts/test.py`, for simple build+test of PyMuPDF git checkout.
  - Added `fitz.pymupdf_version_tuple`, e.g. `(1, 23, 6)`.
  - Restored mistakenly-reverted fix for 2345: Turn off print statements in utils.py
  - Include any trailing `... repeated <N> times...` text in warnings returned by *`mupdf_warnings()`* (rebased only).

Changes in version 1.23.6 (2023-11-06)

- Bug fixes:
  - **Fixed** 2553: Invalid characters in versions >= 1.22
  - **Fixed** 2608: Incorrect utf32 text extraction (high & low surrogates are split)
  - **Fixed** 2710: page.rect and text location wrong / differing from older version
  - **Fixed** 2774: wrong encoding for "?" character when sort=True
  - **Fixed** 2775: fitz_new does not work with python3.10 or earlier
  - **Fixed** 2777: With fitz_new, wrong type for Page.mediabox
- Other:
  - Use MuPDF-1.23.5.
  - Added Document.resolve_names() (rebased implementation only).

Changes in version 1.23.5 (2023-10-11)

- Bug fixes:
  - **Fixed** 2341: Handling negative values in the zoom section for LINK_GOTO in linkDest
  - **Fixed** 2522: Typo in set_layer() - NameError: name 'f' is not defined
  - **Fixed** 2548: Fitz freezes on some PDFs when calling the fitz.Page.get_text_blocks method.
  - **Fixed** 2596: save(garbage=3) breaks get_pixmap() with side effect
  - **Fixed** 2635: "clean=True" makes objects invisible in the pdf

  - **Fixed** 2637: Page.insert_textbox incorrectly handles the last word if it starts a new line
  - **Fixed** 2699: extract paragraph with below table
  - **Fixed** 2703: Wrong fontsize calculation in corner cases ("page.get_texttrace()")
  - **Fixed** 2710: page.rect and text location wrong / differing from older version
  - **Fixed** 2723: When will a Python 3.12 wheel be available?
  - **Fixed** 2730: persistent get_text() formatting
- Other:
  - Use MuPDF-1.23.4.
  - Fix optimisation flags with system installs.
  - Fixed the problem that the clip parameter does not take effect during table recognition
  - Support Pillow mode "RGBa"
  - Support extra word delimiters
  - Support checking valid PDF name objects

Changes in version 1.23.4 (2023-09-26)

- Improved build instructions.
- Fixed Tesseract in rebased implementation.
- Improvements to build/install with system MuPDF.
- Fixed Pyodide builds.
- Fixed rebased bug in _insert_image().
- Bug fixes:
  - **Fixed** 2556: Segmentation fault at calling get_cdrawings(extended=True)
  - **Fixed** 2637: Page.insert_textbox incorrectly handles the last word if it starts a new line
  - **Fixed** 2683: Windows sdist build failure - non-quoting of path and using UNIX which command
  - **Fixed** 2691: Page.get_textpage_ocr() bug in rebased fitz_new version
  - **Fixed** 2692: Page.get_pixmap(clip=Rect()) bug in rebased fitz_new version

Changes in version 1.23.3 (2023-08-31)

- Fixed use of Tesseract for OCR.

Changes in version 1.23.2 (2023-08-28)

- **Fixed** #2613: release 1.23.0 not MacOS-arm64 compatible

Changes in version 1.23.1 (2023-08-24)

- Updated README and package summary description.
- Fixed a problem on some Linux installations with Python-3.10 (and possibly earlier versions) where `import fitz` failed with `ImportError: libcrypt.so.2: cannot open shared object file: No such file or directory`.
- Fixed `incompatible architecture` error on MacOS arm64.
- Fixed installation warning from Poetry about missing entry in wheels' RECORD files.

Changes in version 1.23.0 (2023-08-22)

- Add method *find_tables()* to the *Page* object.

  This allows locating tables on any supported document page, and extracting table content by cell.

- New "rebased" implementation of PyMuPDF.

  The rebased implementation is available as Python module `fitz_new`. It can be used as a drop-in replacement with `import fitz_new as fitz`.

- Python-independent MuPDF libraries are now in a second wheel called `PyMuPDFb` that will be automatically installed by pip.

  This is to save space on pypi.org - a full release only needs one `PyMuPDFb` wheel for each OS.

- Bug fixes:
  - **Fixed** #2542: fitz.utils.scrub AttributeError Annot object has no attribute fileUpd inside
  - **Fixed** #2533: get_texttrace returned a incorrect character bbox
  - **Fixed** #2537: Validation when setting a grouped RadioButton throws a RuntimeError: path to 'V' has indirects
- Other changes:
  - Dropped support for Python-3.7.
  - Fix for wrong page / annot `/Contents` cleaning.

    We need to set `pdf_filter_options::no_update` to zero.
  - Added new function get_tessdata().
  - Cope with problem `/Annot` arrays.

    When copying page annotations in method Document.insert_pdf we previously did not check the validity of members of the `/Annots` array. For faulty members (like null or non-dictionary items) this could cause unnecessary exceptions. This fix implements more checks and skips such array items.
  - Additional annotation type checks.

    We did not previously check for annotation type when getting / setting annotation border properties. This is now checked in accordance with MuPDF.
  - Increase fault tolerance.

    Avoid exceptions in method *insert_pdf()* when source pages contains invalid items in the `/Annots` array.
  - Return empty border dict for applicable annots.

    We previously were returning a non-empty border dictionary even for non-applicable annotation types. We now return the empty dictionary `{}` in these cases. This requires some corresponding changes in the annotation `.update()` method, namely for dashes and border width.
  - Restrict *set_rect* to applicable annot types.

    We were insufficiently excluding non-applicable annotation types from *set_rect()* method. We now let MuPDF catch unsupported annotations and return `False` in these cases.
  - Wrong fontsize computation in `page.get_texttrace()`.

    When computing the font size we were using the final text transformation matrix, where we should have taken `span->trm` instead. This is corrected here.
  - Updates to cope with changes to latest MuPDF.

    `pdf_lookup_anchor()` has been removed.

  - Update fill_textbox to better respect rect.width

    The function norm_words in fill_textbox had a bug in its last loop, appending n+1 characters when actually measuring width of n characters. It led to a bug in fill_texbox when you tried to write a single word mostly composed of "wide" letters (M,m, W, w...), causing the written text to exceed the given rect.

    The fix was just to replace n+1 by n.

  - Add *script_focus* and *script_blur* options to widget.

Changes in version 1.22.5 (2023-06-21)

- This release uses `MuPDF-1.22.2`.
- Bug fixes:
  - **Fixed** #2365: Incorrect dictionary values for type "fs" drawings.
  - **Fixed** #2391: Check box automatically uncheck when we update same checkbox more than 1 times.
  - **Fixed** #2400: Gaps within text of same line not filled with spaces.
  - **Fixed** #2404: Blacklining an image in PDF won't remove underlying content in version 1.22.X.
  - **Fixed** #2430: Incorrectly reducing ref count of Py_None.
  - **Fixed** #2450: Empty fill color and fill opacity for paths with fill and stroke operations with 1.22.*
  - **Fixed** #2462: Error at "get_drawing(extended=True)"
  - **Fixed** #2468: Decode error when trying to get drawings
  - **Fixed** #2710: page.rect and text location wrong / differing from older version
  - **Fixed** #2723: When will a Python 3.12 wheel be available?
- New features:
  - **Changed** Annotations now support "cloudy" borders. The *Annot.border* property has the new item `clouds`, and method *Annot.set_border()* supports the corresponding `clouds` argument.
  - **Changed** Radio button widgets in the same RB group are now consistently updated **if the group is defined in the standard way**.
  - **Added** Support for the `/Locked` key in PDF Optional Content. This array inside the catalog entry `/OCProperties` can now be extracted and set.
  - **Added** Support for new parameter `tessdata` in OCR functions. New function *get_tessdata()* locates the language support folder if Tesseract is installed.

Changes in version 1.22.3 (2023-05-10)

- This release uses `MuPDF-1.22.0`.
- Bug fixes:
  - **Fixed** #2333: Unable to set any of button radio group in form

Changes in version 1.22.2 (2023-04-26)

- This release uses `MuPDF-1.22.0`.
- Bug fixes:
  - **Fixed** #2369: Image extraction bugs with newer versions

Changes in version 1.22.1 (2023-04-18)

- This release uses `MuPDF-1.22.0`.

- Bug fixes:
  - **Fixed** #2345: Turn off print statements in utils.py
  - **Fixed** #2348: extract_image returns an extension "flate" instead of "png"
  - **Fixed** #2350: Can not make widget (checkbox) to read-only by adding flags PDF_FIELD_IS_READ_ONLY
  - **Fixed** #2355: 1.22.0 error when using get_toc (AttributeError: 'SwigPyObject' object has no attribute)

Changes in version 1.22.0 (2023-04-14)

- This release uses `MuPDF-1.22.0`.
- Behavioural changes:
  - Text extraction now includes glyphs that overlap with clip rect; previously they were included only if they were entirely contained within the clip rect.
- Bug fixes:
  - **Fixed** #1763: Interactive(smartform) form PDF calculation not working in pymupdf
  - **Fixed** #1995: RuntimeError: image is too high for a long paged pdf file when trying
  - **Fixed** #2093: Image in pdf changes color after applying redactions
  - **Fixed** #2108: Redaction removing more text than expected
  - **Fixed** #2141: Failed to read JPX header when trying to get blocks
  - **Fixed** #2144: Replace image throws an error
  - **Fixed** #2146: Wrong Handling of Reference Count of "None" Object
  - **Fixed** #2161: Support adding images as pages directly
  - **Fixed** #2168: `page.add_highlight_annot(start=pointa, stop=pointb)` not working
  - **Fixed** #2173: Double free of `Colorspace` used in `Pixmap`
  - **Fixed** #2179: Incorrect documentation for `pixmap.tint_with()`
  - **Fixed** #2208: Pushbutton widget appears as check box
  - **Fixed** #2210: `apply_redactions()` move pdf text to right after redaction
  - **Fixed** #2220: `Page.delete_image()` | object has no attribute `is_image`
  - **Fixed** #2228: open some pdf cost too much time
  - **Fixed** #2238: Bug - can not extract data from file in the newest version 1.21.1
  - **Fixed** #2242: Python quits silently in `Story.element_positions()` if callback function prototype is wrong
  - **Fixed** #2246: TextWriter write text in a wrong position
  - **Fixed** #2248: After redacting the content, the position of the remaining text changes
  - **Fixed** #2250: docs: unclear or broken link in page.rst
  - **Fixed** #2251: mupdf_display_errors does not apply to Pixmap when loading broken image
  - **Fixed** #2270: `Annot.get_text("words")` - doesn't return the first line of words
  - **Fixed** #2275: insert_image: document that rotations are counterclockwise

  - **Fixed** #2278: Can not make widget (checkbox) to read-only by adding flags PDF_FIELD_IS_READ_ONLY
  - **Fixed** #2290: Different image format/data from Page.get_text("dict") and Fitz.get_page_images()
  - **Fixed** #2293: 68 failed tests when installing from sdist on my box
  - **Fixed** #2300: Too much recursion in tree (parents), makes program terminate
  - **Fixed** #2322: add_highlight_annot using clip generates "A Number is Out of Range" error in PDF
- Other:
  - Add key "/AS (Yes)" to the underlying annot object of a selected button form field.
  - Remove unused `Document` methods `has_xref_streams()` and `has_old_style_xrefs()` as MuPDF equivalents have been removed.
  - Add new `Document` methods and properties for getting/setting `/PageMode`, `/PageLayout` and `/MarkInfo`.
  - New `Document` property `version_count`, which contains the number of incremental saves plus one.
  - New `Document` property `is_fast_webaccess` which tells whether the document is linearized.
  - `DocumentWriter` is now a context manager.
  - Add support for `Pixmap` JPEG output.
  - Add support for drawing rectangles with rounded corners.
  - `get_drawings()`: added optional `extended` arg.
  - Fixed issue where trace devices' state was not being initialised correctly; data returned from things like `fitz.Page.get_texttrace()` might be slightly altered, e.g. `linewidth` values.
  - Output warning to `stderr` if it looks like we are being used with current directory containing an invalid `fitz/` directory, because this can break import of `fitz` module. For example this happens if one attempts to use `fitz` when current directory is a PyMuPDF checkout.
- Documentation:
  - General rework:
    - Introduces a new home page and new table of contents.
    - Structural update to include new About section.
    - Comparison & performance graphing.
    - Includes performance methodology in appendix.
    - Updates conf.py to understand single back-ticks as code.
    - Converts double back-ticks to single back-ticks.
    - Removes redundant files.
  - Improve `insert_file()` documentation.
  - `get_bboxlog()`: added optional `layers` to `get_bboxlog()`.
  - `Page.get_texttrace()`: add new dictionary key `layer`, name of Optional Content Group.
  - Mention use of Python venv in installation documentation.
  - Added missing fix for #2057 to release 1.21.1's changelog.
  - Fixes many links to the PyMuPDF-Utilities repo scripts.

  - Avoid duplication of `changes.txt` and `docs/changes.rst`.

- Build
  - Added `pyproject.toml` file to improve builds using pip etc.

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 Response 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 OCRed 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 OCRed 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 OCRed 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 OCRing) 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 enveloping 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 (Arabian, 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 OCRed 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_unicodes()* which return the respective available data.
- **Added** `Page.getDrawings()` which returns details of drawing operations on a document page. Works for all document types.
- Improved performance of *Document.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 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 **permanently modified** by erasing the overlap areas. Also links are removed if overlapped by redactions. This is now fully in sync with PDF specifications.

Other changes:

- **Changed** `TextWriter.writeText()` to support the *"morph"* parameter.
- **Added** methods *Rect.morph()*, *IRect.morph()*, and *Quad.morph()*, which return a new *Quad*.
- **Changed** *Page.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.transformationMattrix` 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.

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 *Form XObject* 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 inserted 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 *Diagnostics* 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 there 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 preceding patch subversions only contained various fixes, this version again introduces major new features:

- **Added** basic support for PDF widget annotations. You can now add PDF form fields of types Text, CheckBox, ListBox and ComboBox. Where necessary, the PDF is transformed to a Form PDF with the first added widget.
- **Fixed** issues #176 ("wrong file embedding"), #177 ("segment fault when invoking page.getText()")and #179 ("Segmentation fault using page.getLinks() on encrypted PDF").

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._setContents()` defines an object given by its *xref* to serve as the *contents* object.
- Changed and extended PDF form field support: the attribute *widget_text* has been renamed to `Annot.widget_value`. Values of all form field types (except signatures) are now supported. A new attribute `Annot.widget_choices` contains the selectable values of listboxes and comboboxes. All these attributes now contain *None* if no value is present.

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 * ~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 amd infinite rectangles and how we handle them. The handling itself was also updated for more consistency in this area.
- We have started basic support for **generation** of PDF content:
  - *Document.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 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 $n < pageCount$. So, e.g. *doc[-500]* is always valid and will load page *(-500) % pageCount*.
- A document can now also be used as an iterator like this: *for page in doc: ...<do something with "page">...*. This will yield all pages of *doc* as *page*.
- The *Pixmap* method *getSize()* has been replaced with property *size*. As before *Pixmap.size == len(Pixmap)* is true.
- In response to transparency (alpha) being optional, several new parameters and properties have been added to *Pixmap* and *Colorspace* classes to support determining their characteristics.
- The *Page* class now contains new properties *firstAnnot* and *firstLink* to provide starting points to the respective class chains, where *firstLink* is just a mnemonic synonym to method *loadLinks()* which continues to exist. Similarly, the new property *rect* is a synonym for method *bound()*, which also continues to exist.
- *Pixmap* methods *samplesRGB()* and *samplesAlpha()* have been deleted because pixmaps can now be created without transparency.
- *Rect* now has a property *irect* which is a synonym of method *round()*. Likewise, *IRect* now has property *rect* to deliver a *Rect* which has the same coordinates as floats values.
- Document has the new method *searchPageFor()* to search for a text string. It works exactly like the corresponding *Page.searchFor()* with page number as additional parameter.

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 colorspace converted to *cspace*. This works for all colorspace combinations.
- Pixmap constructor *fitz.Pixmap(colorspace, width, height, samples)* now allows *samples* to also be *bytes*, not only *bytearray*.

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*
- Pixmaps can now be created directly from files using the new constructor *fitz.Pixmap(filename)*.
- The Pixmap constructor *fitz.Pixmap(image)* has been extended accordingly.
- *fitz.Rect* can now be created with all possible combinations of points and coordinates.
- PyMuPDF classes and methods now all contain __doc__ strings, most of them created by SWIG automatically. While the PyMuPDF documentation certainly is more detailed, this feature should help a lot when programming in Python-aware IDEs.
- A new document method of *getPermits()* returns the permissions associated with the current access to the document (print, edit, annotate, copy), as a Python dictionary.
- The identity matrix *fitz.Identity* is now **immutable**.

- The new document method *select(list)* removes all pages from a document that are not contained in the list. Pages can also be duplicated and re-arranged.
- Various improvements and new members in our demo and examples collections. Perhaps most prominently: *PDF_display* now supports scrolling with the mouse wheel, and there is a new example program *wxTableExtract* which allows to graphically identify and extract table data in documents.
- *fitz.open()* is now an alias of *fitz.Document()*.
- New pixmap method *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.

This documentation covers all versions up to 1.28.2.

CHAPTER

THIRTYNINE

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 renamed 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(pymupdf.Document.pageCount.__doc__)
*** Deprecated and removed in version following 1.19.0 - use 'page_count'. ***
Number of pages.
>>> print(pymupdf.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.

This documentation covers all versions up to 1.28.2.

This documentation covers all versions up to 1.28.2.

INDEX

Symbols

__init__() (*Archive method*), 217
__init__() (*Colorspace method*), 219
__init__() (*Device method*), 483, 484
__init__() (*DisplayList method*), 220
__init__() (*Document method*), 225
__init__() (*DocumentWriter method*), 275
__init__() (*IRect method*), 284, 285
__init__() (*IdentifyHeaders method*), 452
__init__() (*Matrix method*), 293, 294
__init__() (*Pixmap method*), 356–358
__init__() (*Point method*), 370, 371
__init__() (*Quad method*), 373
__init__() (*Rect method*), 378
__init__() (*Shape method*), 383
__init__() (*Story method*), 401
__init__() (*TextWriter method*), 419
__init__() (*TocHeaders method*), 455

A

a (*Matrix attribute*), 295
abs_unit (*Point attribute*), 372
add() (*Archive method*), 218
add_bullet_list() (*Xml method*), 437
add_caret_annot() (*Page method*), 303
add_circle_annot() (*Page method*), 307
add_class() (*Xml method*), 440
add_code() (*Xml method*), 438
add_codeblock() (*Xml method*), 437
add_description_list() (*Xml method*), 437
add_division() (*Xml method*), 437
add_file_annot
    examples, 102
add_file_annot() (*Page method*), 306
add_freetext_annot
    align, 304
    border_width, 304
    callout, 304
    dashes, 304
    fill_color, 304
    fontname, 304
    fontsize, 304
    line_end, 304
    opacity, 304
    rect, 304
    richtext, 304
    rotate, 304
    style, 304
    text_color, 304
add_freetext_annot() (*Page method*), 304
add_header() (*Xml method*), 437
add_highlight_annot() (*Page method*), 310
add_horizontal_line() (*Xml method*), 437
add_image() (*Xml method*), 437
add_ink_annot() (*Page method*), 306
add_kbd() (*Xml method*), 438
add_layer() (*Document method*), 227
add_line_annot() (*Page method*), 306
add_link() (*Xml method*), 437
add_number_list() (*Xml method*), 438
add_ocg() (*Document method*), 228
add_paragraph() (*Xml method*), 438
add_polygon_annot() (*Page method*), 310
add_polyline_annot() (*Page method*), 310
add_rect_annot() (*Page method*), 307
add_redact_annot() (*Page method*), 307
add_samp() (*Xml method*), 438
add_span() (*Xml method*), 438
add_squiggly_annot() (*Page method*), 310
add_stamp_annot() (*Page method*), 316
add_strikeout_annot() (*Page method*), 310
add_style() (*Xml method*), 440
add_subscript() (*Xml method*), 438
add_superscript() (*Xml method*), 438
add_text() (*Xml method*), 438
add_text_annot() (*Page method*), 304
add_underline_annot() (*Page method*), 310
add_var() (*Xml method*), 438
add_widget() (*Page method*), 316
adjust
    insert_htmlbox, 321
adobe_glyph_names(), 467
adobe_glyph_unicodes(), 467
align

add_freetext_annot, 304
insert_textbox, 320, 392
alpha
Annot.get_pixmap, 204
DisplayList.get_pixmap, 221
get_pixmap, 339
alpha (*Pixmap attribute*), 366
Annot (*built-in class*), 204
Annot.get_pixmap
alpha, 204
colorspace, 204
dpi, 204
matrix, 204
Annot.get_text
blocks, 204
clip, 204
dict, 204
flags, 204
html, 204
json, 204
rawdict, 204
text, 204
words, 204
xhtml, 204
xml, 204
Annot.update
blend_mode, 210
border_color, 210
cross_out, 210
fill_color, 210
fontsize, 210
rotate, 210
text_color, 210
Annot.update_file
buffer, 211
desc, 211
filename, 211
ufilename, 211
annot_names() (*Page method*), 341
annot_xrefs() (*Page method*), 341
annots
Document.insert_file, 256
Document.insert_pdf, 255
get_pixmap, 339
annots() (*Page method*), 319
append
Document.insert_file, 256
Document.insert_pdf, 255
append() (*TextWriter method*), 419
append_child() (*Xml method*), 441
appendv() (*TextWriter method*), 420
apply_css() (*Document method*), 269
apply_redactions() (*Page method*), 308
archive
insert_htmlbox, 321
Archive (*built-in class*), 217
artbox (*Page attribute*), 349
ascender (*Font attribute*), 283
attach
embed file, 32, 33
authenticate() (*Document method*), 231

B

b (*Matrix attribute*), 295
bake() (*Document method*), 251
Base14_Fonts (*built-in variable*), 493
bbox (*Font attribute*), 283
bbox (*Table attribute*), 353
bbox (*TableHeader attribute*), 354
begin_page() (*DocumentWriter method*), 276
bl (*IRect attribute*), 286
bl (*Rect attribute*), 381
bleedbox (*Page attribute*), 349
blend_mode
Annot.update, 210
blendmode (*Annot attribute*), 207
blocks
Annot.get_text, 204
Page.get_text, 329
body (*Story attribute*), 403
body_limit (*IdentifyHeaders attribute*), 453
border (*Annot attribute*), 214
border (*Link attribute*), 289
border_color
Annot.update, 210
border_color (*Widget attribute*), 431
border_dashes (*Widget attribute*), 431
border_style (*Widget attribute*), 431
border_width
add_freetext_annot, 304
insert_text, 320, 390
insert_textbox, 320, 392
border_width (*Widget attribute*), 431
bottom_left (*IRect attribute*), 286
bottom_left (*Rect attribute*), 381
bottom_right (*IRect attribute*), 286
bottom_right (*Rect attribute*), 381
bound() (*Page method*), 303
br (*IRect attribute*), 286
br (*Rect attribute*), 381
breadth
draw_squiggle, 323, 383
draw_zigzag, 323, 385
buffer
Annot.update_file, 211
buffer (*Font attribute*), 282
button_caption (*Widget attribute*), 432
button_states() (*Widget method*), 430

C

c (*Matrix attribute*), 295
callout
    add_freetext_annot, 304
can_save_incrementally() (*Document method*), 251
catalog (*built-in variable*), 488
cells (*Table attribute*), 353
cells (*TableHeader attribute*), 354
chapter_count (*Document attribute*), 272
chapter_page_count() (*Document method*), 234
char_lengths() (*Font method*), 281
choice_values (*Widget attribute*), 431
clean_contents() (*Annot method*), 480
clean_contents() (*Page method*), 479
clear_with() (*Pixmap method*), 358
clip
    Annot.get_text, 204
    DisplayList.get_pixmap, 221
    get_pixmap, 339
    get_textpage, 331
    Page.get_text, 329
    search_for, 345
    show_pdf_page, 343
clip_to_rect() (*Page method*), 342
clone() (*Xml method*), 441
close() (*Document method*), 262
close() (*DocumentWriter method*), 276
closePath
    draw_bezier, 324
    draw_circle, 323
    draw_curve, 324
    draw_line, 322
    draw_oval, 323
    draw_polyline, 323
    draw_quad, 324
    draw_rect, 324
    draw_sector, 323
    draw_squiggle, 323
    draw_zigzag, 323
    finish, 389
cluster_drawings() (*Page method*), 312
col_count (*Table attribute*), 353
color
    Document.insert_page, 257
    draw_bezier, 324
    draw_circle, 323
    draw_curve, 324
    draw_line, 322
    draw_oval, 323
    draw_polyline, 323
    draw_quad, 324
    draw_rect, 324
    draw_sector, 323
    draw_squiggle, 323
    draw_zigzag, 323
    finish, 389
    insert_text, 320, 390
    insert_textbox, 320, 392
color (*TextWriter attribute*), 422
color_count() (*Pixmap method*), 365
color_topusage() (*Pixmap method*), 366
colors (*Annot attribute*), 214
colors (*Link attribute*), 289
colors_pdf_dict(), 483
colors_wx_list(), 483
colorspace
    Annot.get_pixmap, 204
    DisplayList.get_pixmap, 221
    get_pixmap, 339
Colorspace (*built-in class*), 219
colorspace (*Pixmap attribute*), 366
commit
    overlay, 393
commit() (*Shape method*), 393
concat() (*Matrix method*), 295
contains() (*IRect method*), 285
contains() (*Rect method*), 379
contents (*built-in variable*), 488
ConversionHeader(), 474
ConversionTrailer(), 474
convert_to_pdf
    examples, 99
convert_to_pdf() (*Document method*), 239
coordinate (*built-in variable*), 487
copy
    examples, 107, 108
copy() (*Pixmap method*), 361
copy_page() (*Document method*), 258
create_element() (*Xml method*), 441
create_text_node() (*Xml method*), 441
CropBox (*built-in variable*), 487
cropbox (*Page attribute*), 348
cropbox_position (*Page attribute*), 348
cross_out
    Annot.update, 210
CS_CMYK (*built-in variable*), 493
CS_GRAY (*built-in variable*), 493
CS_RGB (*built-in variable*), 493
csCMYK (*built-in variable*), 493
csGRAY (*built-in variable*), 493
csRGB (*built-in variable*), 493
css
    insert_htmlbox, 321
css_for_pymupdf_font(), 467

D

d (*Matrix attribute*), 295
dashes

add_freetext_annot, 304
draw_bezier, 324
draw_circle, 323
draw_curve, 324
draw_line, 322
draw_oval, 323
draw_polyline, 323
draw_quad, 324
draw_rect, 324
draw_sector, 323
draw_squiggle, 323
draw_zigzag, 323
finish, 389
debug() (*Xml method*), 441
del_toc_item() (*Document method*), 249
del_xml_metadata() (*Document method*), 474
delete_annot() (*Page method*), 317
delete_image
xref, 328
delete_image() (*Page method*), 328
delete_link() (*Page method*), 317
delete_page() (*Document method*), 257
delete_pages() (*Document method*), 257
delete_responses() (*Annot method*), 210
delete_widget() (*Page method*), 317
delimiters
Page.get_text, 329
derotation_matrix (*Page attribute*), 349
desc
Annot.update_file, 211
Document.embfile_add, 259
Document.embfile_upd, 261
descender (*Font attribute*), 283
dest (*Link attribute*), 290
dest (*linkDest attribute*), 291
dest (*Outline attribute*), 300
Device (*built-in class*), 483
dict
Annot.get_text, 204
Page.get_text, 329
dictionary (*built-in variable*), 488
digest (*Pixmap attribute*), 366
DisplayList (*built-in class*), 220
DisplayList.get_pixmap
alpha, 221
clip, 221
colorspace, 221
matrix, 221
distance_to() (*Point method*), 371
doc (*Shape attribute*), 393
Document
filename, 225
filetype, 225
fontsize, 225
open, 225
rect, 225
stream, 225
Document (*built-in class*), 224
Document.convert_to_pdf
from_page, 239
rotate, 239
to_page, 239
Document.embfile_add
desc, 259
filename, 259
ufilename, 259
Document.embfile_upd
desc, 261
filename, 261
ufilename, 261
Document.insert_file
annots, 256
append, 256
from_page, 256
join, 256
links, 256
merge, 256
rotate, 256
show_progress, 256
start_at, 256
to_page, 256
Document.insert_page
color, 257
fontfile, 257
fontname, 257
fontsize, 257
height, 257
width, 257
Document.insert_pdf
annots, 255
append, 255
from_page, 255
join, 255
join_duplicates, 255
links, 255
merge, 255
rotate, 255
show_progress, 255
start_at, 255
to_page, 255
widgets, 255
Document.layout
fontsize, 247
height, 247
rect, 247
width, 247
Document.new_page
height, 256

    width, 256
DocumentWriter (*built-in class*), 275
down (*Outline attribute*), 299
dpi
    Annot.get_pixmap, 204
    get_pixmap, 339
    get_textpage_ocr, 331
draw() (*Story method*), 402
draw_bezier
    closePath, 324
    color, 324
    dashes, 324
    fill, 324
    fill_opacity, 324
    lineCap, 324
    lineJoin, 324
    morph, 324
    oc, 324
    overlay, 324
    stroke_opacity, 324
    width, 324
draw_bezier() (*Page method*), 324
draw_bezier() (*Shape method*), 385
draw_circle
    closePath, 323
    color, 323
    dashes, 323
    fill, 323
    fill_opacity, 323
    lineCap, 323
    lineJoin, 323
    morph, 323
    oc, 323
    overlay, 323
    stroke_opacity, 323
    width, 323
draw_circle() (*Page method*), 323
draw_circle() (*Shape method*), 387
draw_cont (*Shape attribute*), 394
draw_curve
    closePath, 324
    color, 324
    dashes, 324
    fill, 324
    fill_opacity, 324
    lineCap, 324
    lineJoin, 324
    morph, 324
    oc, 324
    overlay, 324
    stroke_opacity, 324
    width, 324
draw_curve() (*Page method*), 324
draw_curve() (*Shape method*), 387
draw_line
    closePath, 322
    color, 322
    dashes, 322
    fill, 322
    fill_opacity, 322
    lineCap, 322
    lineJoin, 322
    morph, 322
    oc, 322
    overlay, 322
    stroke_opacity, 322
    width, 322
draw_line() (*Page method*), 322
draw_line() (*Shape method*), 383
draw_oval
    closePath, 323
    color, 323
    dashes, 323
    fill, 323
    fill_opacity, 323
    lineCap, 323
    lineJoin, 323
    morph, 323
    oc, 323
    overlay, 323
    stroke_opacity, 323
    width, 323
draw_oval() (*Page method*), 323
draw_oval() (*Shape method*), 386
draw_polyline
    closePath, 323
    color, 323
    dashes, 323
    fill, 323
    fill_opacity, 323
    lineCap, 323
    lineJoin, 323
    morph, 323
    oc, 323
    overlay, 323
    stroke_opacity, 323
    width, 323
draw_polyline() (*Page method*), 323
draw_polyline() (*Shape method*), 385
draw_quad
    closePath, 324
    color, 324
    dashes, 324
    fill, 324
    fill_opacity, 324
    lineCap, 324
    lineJoin, 324
    morph, 324

oc, 324
overlay, 324
stroke_opacity, 324
width, 324
draw_quad() (*Page method*), 324
draw_quad() (*Shape method*), 389
draw_rect
closePath, 324
color, 324
dashes, 324
fill, 324
fill_opacity, 324
lineCap, 324
lineJoin, 324
morph, 324
oc, 324
overlay, 324
radius, 324
stroke_opacity, 324
width, 324
draw_rect() (*Page method*), 324
draw_rect() (*Shape method*), 388
draw_sector
closePath, 323
color, 323
dashes, 323
fill, 323
fill_opacity, 323
fullSector, 323, 388
lineCap, 323
lineJoin, 323
morph, 323
oc, 323
overlay, 323
stroke_opacity, 323
width, 323
draw_sector() (*Page method*), 323
draw_sector() (*Shape method*), 388
draw_squiggle
breadth, 323, 383
closePath, 323
color, 323
dashes, 323
fill, 323
fill_opacity, 323
lineCap, 323
lineJoin, 323
morph, 323
oc, 323
overlay, 323
stroke_opacity, 323
width, 323
draw_squiggle() (*Page method*), 323
draw_squiggle() (*Shape method*), 383
draw_zigzag
breadth, 323, 385
closePath, 323
color, 323
dashes, 323
fill, 323
fill_opacity, 323
lineCap, 323
lineJoin, 323
morph, 323
oc, 323
overlay, 323
stroke_opacity, 323
width, 323
draw_zigzag() (*Page method*), 323
draw_zigzag() (*Shape method*), 385

E

e (*Matrix attribute*), 295
element_positions() (*Story method*), 402
embed
file, attach, 32, 33
PDF, picture, 102
embfile_add
examples, 102, 105
embfile_add() (*Document method*), 259
embfile_count() (*Document method*), 260
embfile_del() (*Document method*), 260
embfile_get() (*Document method*), 260
embfile_info() (*Document method*), 260
embfile_names() (*Document method*), 261
embfile_upd() (*Document method*), 261
EMPTY_IRECT(), 483
EMPTY_QUAD(), 483
EMPTY_RECT(), 483
encoding
insert_font, 324
insert_text, 320, 390
insert_textbox, 320, 392
end_page() (*DocumentWriter method*), 276
entry_list (*Archive attribute*), 219
even_odd
finish, 389
examples
add_file_annot, 102
convert_to_pdf, 99
copy, 107, 108
embfile_add, 102, 105
extract_image, 100
insert_image, 102, 105
invert_irect, 108
JPEG, 105
PhotoImage, 105
Photoshop, 105

Postscript, 105
save, 105, 108
set_rect, 108
show_pdf_page, 102, 105
tobytes, 105
expandtabs
insert_textbox, 320, 392
external (*TableHeader attribute*), 354
extract
image non-PDF, 99
image PDF, 100
table, 81
text rectangle, 80
extract() (*Table method*), 353
extract_font() (*Document method*), 266
extract_image
examples, 100
extract_image() (*Document method*), 264
extractBLOCKS() (*TextPage method*), 408
extractDICT() (*TextPage method*), 409
extractHTML() (*TextPage method*), 409
extractJSON() (*TextPage method*), 409
extractRAWDICT() (*TextPage method*), 410
extractRAWJSON() (*TextPage method*), 410
extractTEXT() (*TextPage method*), 407
extractText() (*TextPage method*), 407
extractWORDS() (*TextPage method*), 408
extractXHTML() (*TextPage method*), 409
extractXML() (*TextPage method*), 409
ez_save() (*Document method*), 254

F

f (*Matrix attribute*), 295
field_flags (*Widget attribute*), 432
field_label (*Widget attribute*), 432
field_name (*Widget attribute*), 432
field_type (*Widget attribute*), 432
field_type_string (*Widget attribute*), 432
field_value (*Widget attribute*), 432
file
attach embed, 32, 33
file extension
wrong, 64
file_info (*Annot attribute*), 211
filename
Annot.update_file, 211
Document, 225
Document.embfile_add, 259
Document.embfile_upd, 261
insert_image, 326
open, 225
replace_image, 328
fileSpec (*linkDest attribute*), 291
filetype
Document, 225
open, 225
fill
draw_bezier, 324
draw_circle, 323
draw_curve, 324
draw_line, 322
draw_oval, 323
draw_polyline, 323
draw_quad, 324
draw_rect, 324
draw_sector, 323
draw_squiggle, 323
draw_zigzag, 323
finish, 389
insert_text, 320, 390
insert_textbox, 320, 392
fill_color
add_freetext_annot, 304
Annot.update, 210
fill_color (*Widget attribute*), 432
fill_opacity
draw_bezier, 324
draw_circle, 323
draw_curve, 324
draw_line, 322
draw_oval, 323
draw_polyline, 323
draw_quad, 324
draw_rect, 324
draw_sector, 323
draw_squiggle, 323
draw_zigzag, 323
finish, 389
insert_text, 320, 390
insert_textbox, 320
fill_textbox() (*TextWriter method*), 421
find() (*Xml method*), 441
find_bookmark() (*Document method*), 234
find_next() (*Xml method*), 442
find_tables() (*Page method*), 312
finish
closePath, 389
color, 389
dashes, 389
even_odd, 389
fill, 389
fill_opacity, 389
lineCap, 389
lineJoin, 389
morph, 389
oc, 389
stroke_opacity, 389
width, 389

finish() (*Shape method*), 389
first_annot (*Page attribute*), 350
first_child (*Xml attribute*), 442
first_link (*Page attribute*), 350
first_widget (*Page attribute*), 350
fit() (*Story method*), 405
fit_height() (*Story method*), 405
fit_scale() (*Story method*), 405
fit_width() (*Story method*), 406
fitz_config (*Tools attribute*), 427
fitz_fontdescriptors, 470
flags
    Annot.get_text, 204
    get_textpage, 331
    get_textpage_ocr, 331
    Page.get_text, 329
    search_for, 345
flags (*Annot attribute*), 213
flags (*Font attribute*), 282
flags (*Link attribute*), 289
flags (*linkDest attribute*), 291
flip_rotate() (*Pixmap method*), 359
Font (*built-in class*), 276
fontbuffer
    insert_font, 324
fontfile
    Document.insert_page, 257
    insert_font, 324
    insert_text, 320, 390
    insert_textbox, 320, 392
fontname
    add_freetext_annot, 304
    Document.insert_page, 257
    insert_font, 324
    insert_text, 320, 390
    insert_textbox, 320, 392
fontsize
    add_freetext_annot, 304
    Annot.update, 210
    Document, 225
    Document.insert_page, 257
    Document.layout, 247
    insert_text, 320, 390
    insert_textbox, 320, 392
    open, 225
fontsize (*built-in variable*), 490
FormFonts (*Document attribute*), 273
from_page
    Document.convert_to_pdf, 239
    Document.insert_file, 256
    Document.insert_pdf, 255
full
    get_textpage_ocr, 331
fullcopy_page() (*Document method*), 258
fullSector
    draw_sector, 323, 388

G

gamma_with() (*Pixmap method*), 359
gen_id() (*Tools method*), 424
get_area() (*IRect method*), 285
get_area() (*Rect method*), 379
get_attribute_value() (*Xml method*), 438
get_attributes() (*Xml method*), 438
get_bboxlog() (*Page method*), 475
get_cdrawings() (*Page method*), 335
get_char_widths() (*Document method*), 480
get_contents() (*Page method*), 479
get_displaylist() (*Page method*), 479
get_drawings() (*Page method*), 332
get_file() (*Annot method*), 211
get_fonts() (*Page method*), 335
get_header_id() (*IdentifyHeaders method*), 453
get_header_id() (*TocHeaders method*), 455
get_image_bbox
    transform, 337
get_image_bbox() (*Page method*), 337
get_image_info
    hashes, 335
    xrefs, 335
get_image_info() (*Page method*), 335
get_image_rects
    transform, 337
get_image_rects() (*Page method*), 337
get_images() (*Page method*), 335
get_key_values(), 452
get_label() (*Page method*), 318
get_layer() (*Document method*), 230
get_layers() (*Document method*), 227
get_links() (*Page method*), 318
get_new_xref() (*Document method*), 481
get_oc() (*Annot method*), 206
get_oc() (*Document method*), 226
get_ocgs() (*Document method*), 230
get_ocmd() (*Document method*), 229
get_page_fonts() (*Document method*), 245
get_page_images() (*Document method*), 245
get_page_labels() (*Document method*), 232
get_page_numbers() (*Document method*), 232
get_page_pixmap() (*Document method*), 244
get_page_text() (*Document method*), 247
get_page_xobjects() (*Document method*), 244
get_pdf_now(), 471
get_pdf_str(), 472
get_pixmap
    alpha, 339
    annots, 339
    clip, 339

colorspace, 339
dpi, 339
matrix, 339
get_pixmap() (*Annot method*), 204
get_pixmap() (*DisplayList method*), 221
get_pixmap() (*Page method*), 339
get_sigflags() (*Document method*), 259
get_sound() (*Annot method*), 211
get_svg_image
matrix, 338
get_svg_image() (*Page method*), 338
get_tessdata(), 482
get_text() (*Annot method*), 204
get_text() (*Page method*), 329
get_text_blocks() (*Page method*), 478
get_text_length(), 471
get_text_words() (*Page method*), 479
get_textbox
rect, 330
textpage, 330
get_textbox() (*Annot method*), 205
get_textbox() (*Page method*), 330
get_textpage
clip, 331
flags, 331
get_textpage() (*Annot method*), 205
get_textpage() (*DisplayList method*), 221
get_textpage() (*Page method*), 331
get_textpage_ocr
dpi, 331
flags, 331
full, 331
language, 331
tessdata, 331
get_textpage_ocr() (*Page method*), 331
get_texttrace() (*Page method*), 475
get_toc() (*Document method*), 240
get_xml_metadata() (*Document method*), 248
get_xobjects() (*Page method*), 337
glyph_advance() (*Font method*), 280
glyph_bbox() (*Font method*), 280
glyph_count (*Font attribute*), 283
glyph_name_to_unicode(), 466
glyph_name_to_unicode() (*Font method*), 280

H

h (*Pixmap attribute*), 368
has_annots() (*Document method*), 267
has_entry() (*Archive method*), 218
has_glyph() (*Font method*), 279
has_links() (*Document method*), 267
has_popup (*Annot attribute*), 214
hashes
get_image_info, 335
header (*Table attribute*), 353
header_id (*IdentifyHeaders attribute*), 453
height
Document.insert_page, 257
Document.layout, 247
Document.new_page, 256
open, 225
height (*IRect attribute*), 287
height (*Pixmap attribute*), 368
height (*Quad attribute*), 375
height (*Rect attribute*), 381
height (*Shape attribute*), 394
html
Annot.get_text, 204
Page.get_text, 329

I

IdentifyHeaders (*built-in class*), 452
image
non-PDF, extract, 99
PDF, extract, 100
resolution, 97
SVG, vector, 105
image_profile(), 472
include_point() (*Rect method*), 379
include_rect() (*Rect method*), 379
INFINITE_IRECT(), 483
INFINITE_QUAD(), 483
INFINITE_RECT(), 483
info (*Annot attribute*), 213
inheritable (*built-in variable*), 487
insert_after() (*Xml method*), 441
insert_before() (*Xml method*), 441
insert_file() (*Document method*), 256
insert_font
encoding, 324
fontbuffer, 324
fontfile, 324
fontname, 324
set_simple, 324
insert_font() (*Page method*), 324
insert_htmlbox
adjust, 321
archive, 321
css, 321
morph, 321
oc, 321
opacity, 321
overlay, 321
rect, 321
rotate, 321
text, 321
insert_htmlbox() (*Page method*), 321
insert_image

examples, 102, 105
filename, 326
keep_proportion, 326
mask, 326
oc, 326
overlay, 326
pixmap, 326
rotate, 326
stream, 326
xref, 326
insert_image() (*Page method*), 326
insert_link() (*Page method*), 317
insert_page() (*Document method*), 257
insert_pdf() (*Document method*), 255
insert_text
border_width, 320, 390
color, 320, 390
encoding, 320, 390
fill, 320, 390
fill_opacity, 320, 390
fontfile, 320, 390
fontname, 320, 390
fontsize, 320, 390
lineheight, 390
miter_limit, 320, 390
morph, 320, 390
oc, 320, 390
overlay, 320
render_mode, 320, 390
rotate, 320, 390
stroke_opacity, 320, 390
insert_text() (*Page method*), 320
insert_text() (*Shape method*), 390
insert_textbox
align, 320, 392
border_width, 320, 392
color, 320, 392
encoding, 320, 392
expandtabs, 320, 392
fill, 320, 392
fill_opacity, 320
fontfile, 320, 392
fontname, 320, 392
fontsize, 320, 392
lineheight, 392
miter_limit, 320, 392
morph, 320, 392
oc, 320, 392
overlay, 320
render_mode, 320, 392
rotate, 320, 392
stroke_opacity, 320
insert_textbox() (*Page method*), 320
insert_textbox() (*Shape method*), 392
interpolate (*Pixmap attribute*), 369
intersect() (*IRect method*), 285
intersect() (*Rect method*), 379
intersects() (*IRect method*), 285
intersects() (*Rect method*), 379
invert() (*Matrix method*), 295
invert_irect
examples, 108
invert_irect() (*Pixmap method*), 361
IRect (*built-in class*), 284
irect (*Pixmap attribute*), 367
irect (*Rect attribute*), 380
irect_like (*built-in variable*), 487
irt_xref (*Annot attribute*), 214
is_bold (*Font attribute*), 283
is_closed (*Document attribute*), 270
is_convex (*Quad attribute*), 375
is_dirty (*Document attribute*), 270
is_empty (*IRect attribute*), 287
is_empty (*Quad attribute*), 375
is_empty (*Rect attribute*), 382
is_encrypted (*Document attribute*), 271
is_external (*Link attribute*), 290
is_external (*Outline attribute*), 300
is_fast_webaccess (*Document attribute*), 270
is_form_pdf (*Document attribute*), 270
is_infinite (*IRect attribute*), 287
is_infinite (*Rect attribute*), 382
is_italic (*Font attribute*), 283
is_monochrome (*Pixmap attribute*), 366
is_monospaced (*Font attribute*), 283
is_open (*Annot attribute*), 214
is_open (*Outline attribute*), 300
is_pdf (*Document attribute*), 270
is_rectangular (*Quad attribute*), 375
is_rectilinear (*Matrix attribute*), 296
is_reflowable (*Document attribute*), 270
is_repaired (*Document attribute*), 270
is_serif (*Font attribute*), 283
is_signed (*Widget attribute*), 432
is_stream() (*Document method*), 481
is_text (*Xml attribute*), 442
is_unicolor (*Pixmap attribute*), 367
is_valid (*Rect attribute*), 382
is_wrapped (*Page attribute*), 478
isMap (*linkDest attribute*), 291
isUri (*linkDest attribute*), 292

J

join
Document.insert_file, 256
Document.insert_pdf, 255
join_duplicates
Document.insert_pdf, 255

journal_can_do() (*Document method*), 268
journal_enable() (*Document method*), 268
journal_load() (*Document method*), 269
journal_op_name() (*Document method*), 268
journal_position() (*Document method*), 268
journal_redo() (*Document method*), 269
journal_save() (*Document method*), 269
journal_start_op() (*Document method*), 268
journal_stop_op() (*Document method*), 268
journal_undo() (*Document method*), 269
JPEG
    examples, 105
json
    Annot.get_text, 204
    Page.get_text, 329

K

keep_proportion
    insert_image, 326
    show_pdf_page, 343
key-value
    lookup text, 79
kind (*linkDest attribute*), 292

L

language
    get_textpage_ocr, 331
last_child (*Xml attribute*), 442
last_location (*Document attribute*), 273
last_point (*TextWriter attribute*), 422
lastPoint (*Shape attribute*), 394
layer_ui_configs() (*Document method*), 231
layout() (*Document method*), 247
ligature (*built-in variable*), 491
line_end
    add_freetext_annot, 304
line_ends (*Annot attribute*), 213
lineCap
    draw_bezier, 324
    draw_circle, 323
    draw_curve, 324
    draw_line, 322
    draw_oval, 323
    draw_polyline, 323
    draw_quad, 324
    draw_rect, 324
    draw_sector, 323
    draw_squiggle, 323
    draw_zigzag, 323
    finish, 389
lineheight
    insert_text, 390
    insert_textbox, 392
lineJoin
    draw_bezier, 324
    draw_circle, 323
    draw_curve, 324
    draw_line, 322
    draw_oval, 323
    draw_polyline, 323
    draw_quad, 324
    draw_rect, 324
    draw_sector, 323
    draw_squiggle, 323
    draw_zigzag, 323
    finish, 389
Link (*built-in class*), 288
LINK_FLAG_B_VALID (*built-in variable*), 499
LINK_FLAG_FIT_H (*built-in variable*), 500
LINK_FLAG_FIT_V (*built-in variable*), 500
LINK_FLAG_L_VALID (*built-in variable*), 499
LINK_FLAG_R_IS_ZOOM (*built-in variable*), 500
LINK_FLAG_R_VALID (*built-in variable*), 499
LINK_FLAG_T_VALID (*built-in variable*), 499
LINK_GOTO (*built-in variable*), 498
LINK_GOTOR (*built-in variable*), 499
LINK_LAUNCH (*built-in variable*), 499
LINK_NAMED (*built-in variable*), 499
LINK_NONE (*built-in variable*), 498
LINK_URI (*built-in variable*), 498
linkDest (*built-in class*), 291
links
    Document.insert_file, 256
    Document.insert_pdf, 255
links() (*Page method*), 318
ll (*Quad attribute*), 374
LlamaMarkdownReader(), 452
load_annot() (*Page method*), 341
load_data() (*pdf_markdown_reader.PDFMarkdownReader method*), 456
load_links() (*Page method*), 342
load_page() (*Document method*), 235
load_widget() (*Page method*), 341
lookup
    text key-value, 79
lr (*Quad attribute*), 374
lt (*linkDest attribute*), 292

M

make_bookmark() (*Document method*), 233
make_table(), 468
markinfo (*Document attribute*), 271
mask
    insert_image, 326
matrix
    Annot.get_pixmap, 204
    DisplayList.get_pixmap, 221
    get_pixmap, 339

    get_svg_image, 338
Matrix (*built-in class*), 293
matrix_like (*built-in variable*), 487
MediaBox (*built-in variable*), 487
mediabox (*Page attribute*), 349
mediabox_size (*Page attribute*), 349
merge
    Document.insert_file, 256
    Document.insert_pdf, 255
metadata (*Document attribute*), 272
miter_limit
    insert_text, 320, 390
    insert_textbox, 320, 392
morph
    draw_bezier, 324
    draw_circle, 323
    draw_curve, 324
    draw_line, 322
    draw_oval, 323
    draw_polyline, 323
    draw_quad, 324
    draw_rect, 324
    draw_sector, 323
    draw_squiggle, 323
    draw_zigzag, 323
    finish, 389
    insert_htmlbox, 321
    insert_text, 320, 390
    insert_textbox, 320, 392
morph() (*IRect method*), 286
morph() (*Quad method*), 373
morph() (*Rect method*), 380
move_page() (*Document method*), 259
mupdf_display_errors() (*Tools method*), 427
mupdf_display_warnings() (*Tools method*), 427
mupdf_version (*built-in variable*), 494
mupdf_version_tuple (*built-in variable*), 494
mupdf_warnings() (*Tools method*), 427

N

n (*Colorspace attribute*), 220
n (*Pixmap attribute*), 369
name (*Colorspace attribute*), 219
name (*Document attribute*), 272
name (*Font attribute*), 282
named (*linkDest attribute*), 292
names (*TableHeader attribute*), 354
need_appearances() (*Document method*), 259
needs_pass (*Document attribute*), 271
new_page() (*Document method*), 256
new_shape() (*Page method*), 345
newWindow (*linkDest attribute*), 292
next (*Annot attribute*), 212
next (*Link attribute*), 290
next (*Outline attribute*), 299
next (*Widget attribute*), 431
next (*Xml attribute*), 442
next_location() (*Document method*), 234
non-PDF
    extract image, 99
norm() (*IRect method*), 286
norm() (*Matrix method*), 294
norm() (*Point method*), 371
norm() (*Rect method*), 380
normalize() (*IRect method*), 286
normalize() (*Rect method*), 380
number (*Page attribute*), 350

O

object (*built-in variable*), 489
oc
    draw_bezier, 324
    draw_circle, 323
    draw_curve, 324
    draw_line, 322
    draw_oval, 323
    draw_polyline, 323
    draw_quad, 324
    draw_rect, 324
    draw_sector, 323
    draw_squiggle, 323
    draw_zigzag, 323
    finish, 389
    insert_htmlbox, 321
    insert_image, 326
    insert_text, 320, 390
    insert_textbox, 320, 392
OCCD (*built-in variable*), 490
OCG (*built-in variable*), 490
OCMD (*built-in variable*), 490
OCPD (*built-in variable*), 490
office_to_json(), 20
office_to_markdown(), 20
office_to_pdf(), 20
on_state() (*Widget method*), 430
opacity
    add_freetext_annot, 304
    insert_htmlbox, 321
opacity (*Annot attribute*), 212
opacity (*TextWriter attribute*), 422
open
    Document, 225
    filename, 225
    filetype, 225
    fontsize, 225
    height, 225
    rect, 225
    stream, 225

width, 225
open(), 65
Outline (*built-in class*), 299
outline (*Document attribute*), 270
outline_xref() (*Document method*), 249
overlay
commit, 393
draw_bezier, 324
draw_circle, 323
draw_curve, 324
draw_line, 322
draw_oval, 323
draw_polyline, 323
draw_quad, 324
draw_rect, 324
draw_sector, 323
draw_squiggle, 323
draw_zigzag, 323
insert_htmlbox, 321
insert_image, 326
insert_text, 320
insert_textbox, 320
show_pdf_page, 343

P

Page (*built-in class*), 303
page (*built-in variable*), 489
page (*linkDest attribute*), 292
page (*Outline attribute*), 299
page (*Shape attribute*), 393
page (*Table attribute*), 353
page (*TableFinder attribute*), 352
Page.get_text
blocks, 329
clip, 329
delimiters, 329
dict, 329
flags, 329
html, 329
json, 329
rawdict, 329
sort, 329
text, 329
textpage, 329
words, 329
xhtml, 329
xml, 329
page_count (*Document attribute*), 272
page_cropbox() (*Document method*), 238
page_xref() (*Document method*), 238
pagelayout (*Document attribute*), 271
pagemode (*Document attribute*), 271
pages() (*Document method*), 239
pagetree (*built-in variable*), 489
paper_rect(), 465
paper_size(), 464
paper_sizes(), 470
parent (*Annot attribute*), 212
parent (*Page attribute*), 350
Partial Pixmaps, 98
PDF
extract image, 100
picture embed, 102
pdf_catalog() (*Document method*), 262
pdf_markdown_reader.PDFMarkdownReader (*built-in class*), 456
pdf_trailer() (*Document method*), 262
pdfcolor, 471
pdfocr_save() (*Pixmap method*), 363
pdfocr_tobytes() (*Pixmap method*), 363
permissions (*Document attribute*), 271
PhotoImage
examples, 105
Photoshop
examples, 105
picture
embed PDF, 102
pil_image() (*Pixmap method*), 364
pil_save() (*Pixmap method*), 364
pil_tobytes() (*Pixmap method*), 364
pixel() (*Pixmap method*), 359
pixmap
insert_image, 326
replace_image, 328
Pixmap (*built-in class*), 356
place() (*Story method*), 402
planish_line(), 469
Point (*built-in class*), 370
point_like (*built-in variable*), 487
popup_rect (*Annot attribute*), 214
popup_xref (*Annot attribute*), 214
Postscript
examples, 105
prerotate() (*Matrix method*), 294
prescale() (*Matrix method*), 294
preshear() (*Matrix method*), 294
pretranslate() (*Matrix method*), 294
prev_location() (*Document method*), 234
previous (*Xml attribute*), 442
pymupdf_date (*built-in variable*), 494
PYMUPDF_MESSAGE, 470
pymupdf_version (*built-in variable*), 494
pymupdf_version_tuple (*built-in variable*), 494

Q

Quad (*built-in class*), 373
quad (*IRect attribute*), 287
quad (*Rect attribute*), 381

quad_like (*built-in variable*), 487
quads
    search_for, 345

R

radius
    draw_rect, 324
rawdict
    Annot.get_text, 204
    Page.get_text, 329
rb (*linkDest attribute*), 292
read_contents() (*Page method*), 480
read_entry() (*Archive method*), 218
reading order
    text, 80
recolor() (*Document method*), 237
recolor() (*Page method*), 342
recover_char_quad(), 482
recover_line_quad(), 482
recover_quad(), 481
recover_span_quad(), 482
rect
    add_freetext_annot, 304
    Document, 225
    Document.layout, 247
    get_textbox, 330
    insert_htmlbox, 321
    open, 225
rect (*Annot attribute*), 212
Rect (*built-in class*), 378
rect (*DisplayList attribute*), 221
rect (*IRect attribute*), 287
rect (*Link attribute*), 290
rect (*Page attribute*), 350
rect (*Quad attribute*), 374
rect (*Shape attribute*), 394
rect (*TextPage attribute*), 411
rect (*TextWriter attribute*), 423
rect (*Widget attribute*), 432
rect_delta (*Annot attribute*), 214
rect_like (*built-in variable*), 487
rectangle
    extract text, 80
reload_page() (*Document method*), 237
remove() (*Xml method*), 441
remove_attribute() (*Xml method*), 439
remove_rotation() (*Page method*), 343
render_mode
    insert_text, 320, 390
    insert_textbox, 320, 392
repair() (*Document method*), 251
replace_image
    filename, 328
    pixmap, 328
    stream, 328
    xref, 328
replace_image() (*Page method*), 328
reset() (*Story method*), 403
reset() (*Widget method*), 431
reset_mupdf_warnings() (*Tools method*), 427
resolution
    image, 97
    zoom, 97
resolution (*built-in variable*), 490
resolve_names() (*Document method*), 238
resources (*built-in variable*), 488
rewrite_images() (*Document method*), 235
richtext
    add_freetext_annot, 304
root (*Xml attribute*), 442
rotate
    add_freetext_annot, 304
    Annot.update, 210
    Document.convert_to_pdf, 239
    Document.insert_file, 256
    Document.insert_pdf, 255
    insert_htmlbox, 321
    insert_image, 326
    insert_text, 320, 390
    insert_textbox, 320, 392
    set_rotation, 342
    show_pdf_page, 343
rotation (*Annot attribute*), 212
rotation (*Page attribute*), 348
rotation_matrix (*Page attribute*), 349
round() (*Rect method*), 378
row_count (*Table attribute*), 353
rows (*Table attribute*), 353
run() (*DisplayList method*), 220
run() (*Page method*), 475

S

samples (*Pixmap attribute*), 367
samples_mv (*Pixmap attribute*), 367
samples_ptr (*Pixmap attribute*), 368
save
    examples, 105, 108
save() (*Document method*), 252
save() (*Pixmap method*), 362
save_snapshot() (*Document method*), 269
saveIncr() (*Document method*), 254
script (*Widget attribute*), 433
script_blur (*Widget attribute*), 433
script_calc (*Widget attribute*), 433
script_change (*Widget attribute*), 433
script_focus (*Widget attribute*), 433
script_format (*Widget attribute*), 433
script_stroke (*Widget attribute*), 433

scrub() (*Document method*), 251
search() (*TextPage method*), 410
search_for
    clip, 345
    flags, 345
    quads, 345
    textpage, 345
search_for() (*Page method*), 345
search_page_for() (*Document method*), 255
select() (*Document method*), 247
set_aa_level() (*Tools method*), 426
set_align() (*Xml method*), 438
set_alpha() (*Pixmap method*), 361
set_annot_stem() (*Tools method*), 424
set_artbox() (*Page method*), 348
set_attribute() (*Xml method*), 438
set_bgcolor() (*Xml method*), 439
set_bleedbox() (*Page method*), 348
set_blendmode() (*Annot method*), 208
set_bold() (*Xml method*), 439
set_border() (*Annot method*), 209
set_border() (*Link method*), 288
set_color() (*Xml method*), 439
set_colors() (*Annot method*), 209
set_colors() (*Link method*), 289
set_columns() (*Xml method*), 439
set_contents() (*Page method*), 479
set_cropbox() (*Page method*), 347
set_dpi() (*Pixmap method*), 360
set_flags() (*Annot method*), 209
set_flags() (*Link method*), 289
set_font() (*Xml method*), 439
set_fontsize() (*Xml method*), 439
set_id() (*Xml method*), 439
set_info() (*Annot method*), 205
set_irt_xref() (*Annot method*), 206
set_italic() (*Xml method*), 439
set_layer() (*Document method*), 230
set_layer_ui_config() (*Document method*), 231
set_leading() (*Xml method*), 440
set_line_ends() (*Annot method*), 206
set_lineheight() (*Xml method*), 440
set_margins() (*Xml method*), 440
set_markinfo() (*Document method*), 248
set_mediabox() (*Page method*), 347
set_messages(), 465
set_metadata() (*Document method*), 247
set_name() (*Annot method*), 208
set_oc() (*Annot method*), 206
set_oc() (*Document method*), 227
set_ocmd() (*Document method*), 228
set_opacity() (*Annot method*), 207
set_open() (*Annot method*), 207
set_origin() (*Pixmap method*), 360
set_page_labels() (*Document method*), 233
set_pagebreak_after() (*Xml method*), 440
set_pagebreak_before() (*Xml method*), 440
set_pagelayout() (*Document method*), 248
set_pagemode() (*Document method*), 248
set_pixel() (*Pixmap method*), 360
set_popup() (*Annot method*), 207
set_properties() (*Xml method*), 440
set_rect
    examples, 108
set_rect() (*Annot method*), 208
set_rect() (*Pixmap method*), 360
set_rotation
    rotate, 342
set_rotation() (*Annot method*), 209
set_rotation() (*Page method*), 342
set_simple
    insert_font, 324
set_small_glyph_heights() (*Tools method*), 425
set_subset_fontnames() (*Tools method*), 425
set_text_indent() (*Xml method*), 440
set_toc() (*Document method*), 248
set_toc_item() (*Document method*), 250
set_trimbox() (*Page method*), 348
set_xml_metadata() (*Document method*), 248
Shape (*built-in class*), 383
show_aa_level() (*Tools method*), 426
show_pdf_page
    clip, 343
    examples, 102, 105
    keep_proportion, 343
    overlay, 343
    rotate, 343
show_pdf_page() (*Page method*), 343
show_progress
    Document.insert_file, 256
    Document.insert_pdf, 255
shrink() (*Pixmap method*), 359
size (*Pixmap attribute*), 368
sort
    Page.get_text, 329
sRGB_to_pdf(), 465
sRGB_to_rgb(), 466
start_at
    Document.insert_file, 256
    Document.insert_pdf, 255
store_maxsize (*Tools attribute*), 429
store_shrink() (*Tools method*), 426
store_size (*Tools attribute*), 429
Story (*built-in class*), 401
Story.FitResult (*built-in class*), 404
stream
    Document, 225
    insert_image, 326

open, 225
replace_image, 328
stream (*built-in variable*), 489
stride (*Pixmap attribute*), 366
stroke_opacity
draw_bezier, 324
draw_circle, 323
draw_curve, 324
draw_line, 322
draw_oval, 323
draw_polyline, 323
draw_quad, 324
draw_rect, 324
draw_sector, 323
draw_squiggle, 323
draw_zigzag, 323
finish, 389
insert_text, 320, 390
insert_textbox, 320
style
add_freetext_annot, 304
subset_fonts() (*Document method*), 267
SVG
vector image, 105
switch_layer() (*Document method*), 227

T

table
extract, 81
Table (*built-in class*), 353
TableFinder (*built-in class*), 352
TableHeader (*built-in class*), 354
TableRow (*built-in class*), 354
tables (*TableFinder attribute*), 352
tagname (*Xml attribute*), 442
tessdata
get_textpage_ocr, 331
text
Annot.get_text, 204
insert_htmlbox, 321
key-value, lookup, 79
Page.get_text, 329
reading order, 80
rectangle, extract, 80
text (*Xml attribute*), 442
TEXT_ACCURATE_BBOXES (*built-in variable*), 497
TEXT_ALIGN_CENTER (*built-in variable*), 495
TEXT_ALIGN_JUSTIFY (*built-in variable*), 495
TEXT_ALIGN_LEFT (*built-in variable*), 495
TEXT_ALIGN_RIGHT (*built-in variable*), 495
TEXT_COLLECT_STRUCTURE (*built-in variable*), 497
TEXT_COLLECT_STYLES (*built-in variable*), 497
TEXT_COLLECT_VECTORS (*built-in variable*), 497
text_color
add_freetext_annot, 304
Annot.update, 210
text_color (*Widget attribute*), 432
text_cont (*Shape attribute*), 394
TEXT_DEHYPHENATE (*built-in variable*), 496
text_font (*Widget attribute*), 432
TEXT_FONT_BOLD (*built-in variable*), 496
TEXT_FONT_ITALIC (*built-in variable*), 496
TEXT_FONT_MONOSPACED (*built-in variable*), 496
TEXT_FONT_SERIFED (*built-in variable*), 496
TEXT_FONT_SUPERSCRIPT (*built-in variable*), 496
text_fontsize (*Widget attribute*), 432
TEXT_FUZZY_VECTORS (*built-in variable*), 497
TEXT_IGNORE_ACTUALTEXT (*built-in variable*), 497
TEXT_INHIBIT_SPACES (*built-in variable*), 496
TEXT_LAZY_VECTORS (*built-in variable*), 497
text_length() (*Font method*), 281
text_maxlen (*Widget attribute*), 432
TEXT_PRESERVE_IMAGES (*built-in variable*), 496
TEXT_PRESERVE_LIGATURES (*built-in variable*), 496
TEXT_PRESERVE_SPANS (*built-in variable*), 497
TEXT_PRESERVE_WHITESPACE (*built-in variable*), 496
text_rect (*TextWriter attribute*), 422
TEXT_SEGMENT (*built-in variable*), 497
text_type (*Widget attribute*), 433
TEXT_USE_CID_FOR_UNKNOWN_UNICODE (*built-in variable*), 497
TEXTFLAGS_BLOCKS (*built-in variable*), 498
TEXTFLAGS_DICT (*built-in variable*), 498
TEXTFLAGS_HTML (*built-in variable*), 498
TEXTFLAGS_RAWDICT (*built-in variable*), 498
TEXTFLAGS_SEARCH (*built-in variable*), 498
TEXTFLAGS_TEXT (*built-in variable*), 498
TEXTFLAGS_WORDS (*built-in variable*), 498
TEXTFLAGS_XHTML (*built-in variable*), 498
TEXTFLAGS_XML (*built-in variable*), 498
textpage
get_textbox, 330
Page.get_text, 329
search_for, 345
TextPage (*built-in class*), 407
TextWriter (*built-in class*), 419
tint_with() (*Pixmap method*), 358
title (*Outline attribute*), 300
tl (*IRect attribute*), 286
tl (*Rect attribute*), 380
to_json(), 450
to_markdown(), 445
to_markdown() (*Table method*), 353
to_page
Document.convert_to_pdf, 239
Document.insert_file, 256
Document.insert_pdf, 255
to_pandas() (*Table method*), 354

to_text(), 449
tobytes
    examples, 105
tobytes() (*Document method*), 254
tobytes() (*Pixmap method*), 362
TocHeaders (*built-in class*), 455
Tools (*built-in class*), 424
top_left (*IRect attribute*), 286
top_left (*Rect attribute*), 380
top_right (*IRect attribute*), 286
top_right (*Rect attribute*), 380
torect() (*IRect method*), 285
torect() (*Rect method*), 380
totalcont (*Shape attribute*), 394
tr (*IRect attribute*), 286
tr (*Rect attribute*), 380
trailer (*built-in variable*), 488
transform
    get_image_bbox, 337
    get_image_rects, 337
transform() (*Point method*), 371
transform() (*Quad method*), 373
transform() (*Rect method*), 378
transformation_matrix (*Page attribute*), 349
trimbox (*Page attribute*), 349
type (*Annot attribute*), 212

U

ufilename
    Annot.update_file, 211
    Document.embfile_add, 259
    Document.embfile_upd, 261
ul (*Quad attribute*), 374
unicode_to_glyph_name(), 466
unicode_to_glyph_name() (*Font method*), 280
unit (*Point attribute*), 371
unitvector (*built-in variable*), 490
unset_quad_corrections() (*Tools method*), 426
update() (*Annot method*), 210
update() (*Widget method*), 431
update_file() (*Annot method*), 211
update_link() (*Page method*), 317
update_object() (*Document method*), 263
update_stream() (*Document method*), 263
ur (*Quad attribute*), 374
uri (*Link attribute*), 290
uri (*linkDest attribute*), 292
uri (*Outline attribute*), 300
use_layout(), 452

V

valid_codepoints() (*Font method*), 279
vector
    image SVG, 105
version, 445
version (*built-in variable*), 494
version_count (*Document attribute*), 271
VersionBind (*built-in variable*), 494
VersionDate (*built-in variable*), 494
VersionFitz (*built-in variable*), 494
vertices (*Annot attribute*), 213

W

w (*Pixmap attribute*), 368
warp() (*Pixmap method*), 365
Widget (*built-in class*), 430
widgets
    Document.insert_pdf, 255
widgets() (*Page method*), 319
width
    Document.insert_page, 257
    Document.layout, 247
    Document.new_page, 256
    draw_bezier, 324
    draw_circle, 323
    draw_curve, 324
    draw_line, 322
    draw_oval, 323
    draw_polyline, 323
    draw_quad, 324
    draw_rect, 324
    draw_sector, 323
    draw_squiggle, 323
    draw_zigzag, 323
    finish, 389
    open, 225
width (*IRect attribute*), 287
width (*Pixmap attribute*), 368
width (*Quad attribute*), 375
width (*Rect attribute*), 381
width (*Shape attribute*), 394
words
    Annot.get_text, 204
    Page.get_text, 329
wrap_contents() (*Page method*), 478
write() (*Story method*), 403
write_stabilized() (*Story static method*), 403
write_stabilized_with_links() (*Story static method*), 404
write_text() (*Page method*), 319
write_text() (*TextWriter method*), 422
write_with_links() (*Story method*), 404
wrong
    file extension, 64

X

x (*Pixmap attribute*), 368
x (*Point attribute*), 372

x0 (*IRect attribute*), 287
x0 (*Rect attribute*), 381
x1 (*IRect attribute*), 287
x1 (*Rect attribute*), 381
xhtml
    Annot.get_text, 204
    Page.get_text, 329
xml
    Annot.get_text, 204
    Page.get_text, 329
Xml (*built-in class*), 437
xml_metadata_xref() (*Document method*), 474
xref
    delete_image, 328
    insert_image, 326
    replace_image, 328
xref (*Annot attribute*), 214
xref (*built-in variable*), 490
xref (*Link attribute*), 290
xref (*Page attribute*), 350
xref (*Widget attribute*), 433
xref_copy() (*Document method*), 264
xref_get_key() (*Document method*), 242
xref_get_keys() (*Document method*), 241
xref_length() (*Document method*), 481
xref_object() (*Document method*), 262
xref_set_key() (*Document method*), 243
xref_stream() (*Document method*), 262
xref_stream_raw() (*Document method*), 262
xref_xml_metadata() (*Document method*), 267
xrefs
    get_image_info, 335
xres (*Pixmap attribute*), 369

Y

y (*Pixmap attribute*), 368
y (*Point attribute*), 372
y0 (*IRect attribute*), 287
y0 (*Rect attribute*), 381
y1 (*IRect attribute*), 287
y1 (*Rect attribute*), 381
yres (*Pixmap attribute*), 369

Z

zoom, 97
    resolution, 97