Collection of Recipes

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

Images

How to Make Images from Document Pages

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

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

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

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

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

How to Increase Image Resolution

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

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

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

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

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

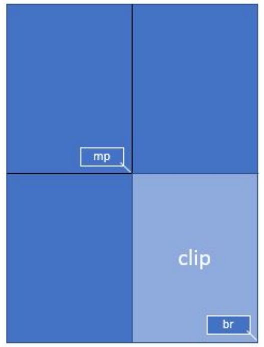
How to Create Partial Pixmaps (Clips)

Table of Contents

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

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

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



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

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

How to Create or Suppress Annotation Images

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

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

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

How to Extract Images: Non-PDF Documents

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

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

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

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

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

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

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

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

How to Extract Images: PDF Documents

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

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

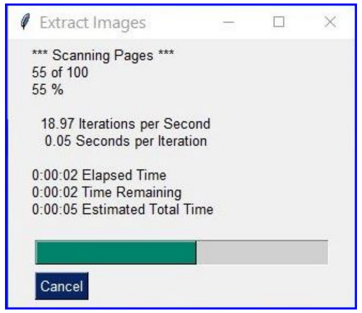
The question remains: **"How do I know those 'xref' numbers of images?"**. There are two answers to this:

a. **"Inspect the page objects:"** Loop through the items of `Page.getImageList()`. It is a list of list, and its items look like *[xref, smask, …]*, containing the `xref` of an image. This `xref` can then be used with one of the above methods. Use this method for **valid (undamaged)** documents. Be wary however, that the same image may be referenced multiple times (by different pages), so you might want to provide a mechanism avoiding multiple extracts.

b. **"No need to know:"** Loop through the list of **all xrefs** of the document and perform a `Document.extractImage()` for each one. If the returned dictionary is empty, then continue – this `xref` is no image. Use this method if the PDF is **damaged (unusable pages)**. Note that a PDF often contains "pseudo-images" ("stencil masks") with the special purpose of defining the transparency of some other image. You may want to provide logic to exclude those from extraction. Also have a look at the next section.

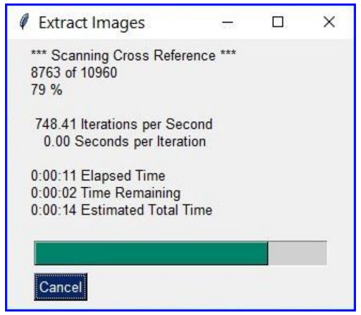
For both extraction approaches, there exist ready-to-use general purpose scripts:

extract-imga.py extracts images page by page:



```
*** Scanning Pages ***
55 of 100
55 %

 18.97 Iterations per Second
  0.05 Seconds per Iteration

0:00:02 Elapsed Time
0:00:02 Time Remaining
0:00:05 Estimated Total Time
```

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



```
*** Scanning Cross Reference ***
8763 of 10960
79 %

 748.41 Iterations per Second
   0.00 Seconds per Iteration

0:00:11 Elapsed Time
0:00:02 Time Remaining
0:00:14 Estimated Total Time
```

How to Handle Stencil Masks

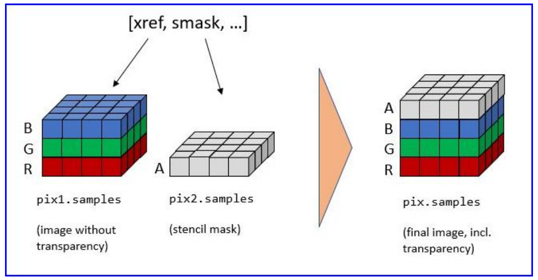
Some images in PDFs are accompanied by **stencil masks**. In their simplest form stencil masks represent alpha (transparency) bytes stored as seperate images. In order to reconstruct the original of an image, which has a stencil mask, it must be "enriched" with transparency bytes taken from its stencil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

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

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

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

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



::

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

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

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

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

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

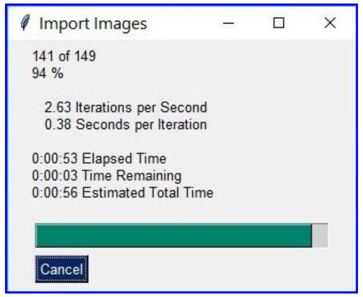
Method 1: Inserting Images as Pages

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

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

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



```
141 of 149
94 %

  2.63 Iterations per Second
  0.38 Seconds per Iteration

0:00:53 Elapsed Time
0:00:03 Time Remaining
0:00:56 Estimated Total Time
```

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

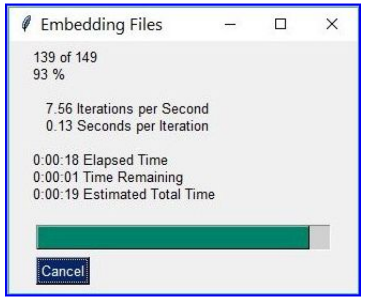
Note

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

Method 2: Embedding Files

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

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

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

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

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

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


Embedding Files

139 of 149
93 %

7.56 Iterations per Second
0.13 Seconds per Iteration

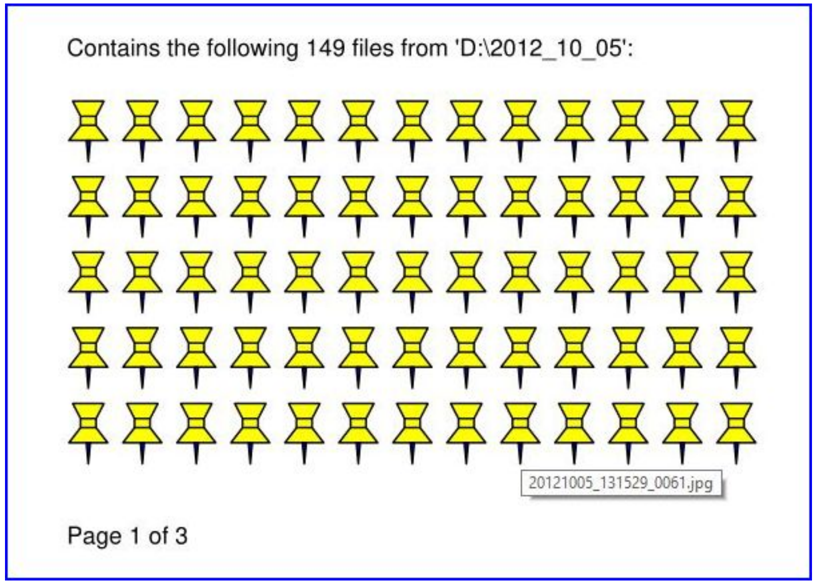
0:00:18 Elapsed Time
0:00:01 Time Remaining
0:00:19 Estimated Total Time

This is by far the fastest method, and it also produces the smallest possible output file size. The above pictures needed 20 seonds on my machine and yielded a PDF size of 510 MB. Look here for a more complete source code: it offers a direcory 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!

How to Create Vector Images

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

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

Instruction *svg = page.getSVGimage(matrix = fitz.Identity)* delivers a UTF-8 string *svg* which can be stored with extension ".svg".

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

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

The general scheme is just the following two lines:

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

Remarks

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

Note

Convert JPEG to Photoshop:

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

Note

Save to JPEG using PIL/Pillow:

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

Note

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

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

Note

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

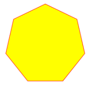
```
if str is bytes:  # this is Python 2!
    import Tkinter as tk
else:  # Python 3 or later!
    import tkinter as tk
pix = fitz.Pixmap("input.png")  # may have an alpha channel
if pix.alpha:  # we have an alpha channel!
    pix = fitz.Pixmap(pix, 0)  # remove it
tkimg = tk.PhotoImage(data=pix.getImageData("ppm"))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

How to Use Pixmaps: Glueing Images

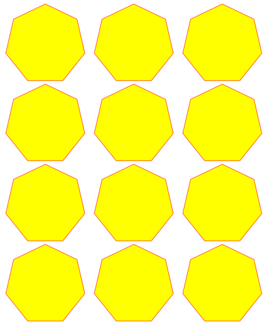
This shows how pixmaps can be used for purely graphical, non-document purposes. The script reads an image file and creates a new image which consist of 3 * 4 tiles of the original:

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

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

# now fill target with the tiles
for i in range(col):
    src.x = src.width * i                # modify input's x coord
    for j in range(lin):
        src.y = src.height * j           # modify input's y coord
        tar_pix.copyPixmap(src, src.irect) # copy input to new loc

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

This is the input picture:



Here is the output:



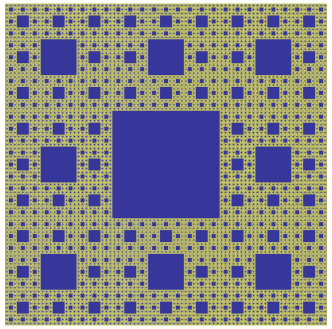
How to Use Pixmaps: Making a Fractal

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 a approximative PNG image of it, by going down to one-pixel granularity. To increase the image precision, change the value of n (precision):

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

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

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

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

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

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

    Arguments are top-left coords and the step width.

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

    return

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

The result should look something like this:



How to Interface with NumPy

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

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

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

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

How to Add Images to a PDF Page

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

Criterion	`Page.insertImage()`	`Page.showPDFpage()`
displayable content	image file, image in memory, pixmap	PDF page
display resolution	image resolution	vectorized (except raster page content)
rotation	multiple of 90 degrees	any angle
clipping	no (full image only)	yes
keep aspect ratio	yes (default option)	yes (default option)
transparency (water marking)	depends on image	yes
location / placement	scaled to fit target rectangle	scaled to fit target rectangle
performance	automatic prevention of duplicates; MD5 calculation on every execution	automatic prevention of duplicates; faster than `Page.insertImage()`

Criterion	Page.insertImage()	Page.showPDFpage()
multi-page image support	no	yes
ease of use	simple, intuitive; performance considerations apply for multiple insertions of same image	simple, intuitive; **usable for all document types** (including images!) after conversion to PDF via Document.convertToPDF()

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

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

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

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

Text

How to Extract all Document Text

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

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

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

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

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

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

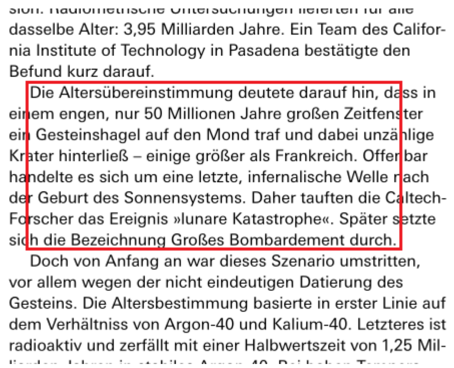
1. Extract text in HTML format and store it as a HTML document, so it can be viewed in any browser.
2. Extract text as a list of text blocks via *Page.getText("blocks")*. Each item of this list contains position information for its text, which can be used to establish a convenient reading order.
3. Extract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

See the following two section for examples and further explanations.

How to Extract Text from within a Rectangle

Please refer to the script textboxtract.py.

It demonstrates ways to extract text contained in the following red rectangle,



sion. Radiometrische Untersuchungen lieferten für alle
dasselbe Alter: 3,95 Milliarden Jahre. Ein Team des Califor-
nia Institute of Technology in Pasadena bestätigte den
Befund kurz darauf.
Die Altersübereinstimmung deutete darauf hin, dass in
einem engen, nur 50 Millionen Jahre großen Zeitfenster
ein Gesteinshagel auf den Mond traf und dabei unzählige
Krater hinterließ – einige größer als Frankreich. Offenbar
handelte es sich um eine letzte, infernalische Welle nach
der Geburt des Sonnensystems. Daher tauften die Caltech-
Forscher das Ereignis »lunare Katastrophe«. Später setzte
sich die Bezeichnung Großes Bombardement durch.
Doch von Anfang an war dieses Szenario umstritten,
vor allem wegen der nicht eindeutigen Datierung des
Gesteins. Die Altersbestimmung basierte in erster Linie auf
dem Verhältniss von Argon-40 und Kalium-40. Letzteres ist
radioaktiv und zerfällt mit einer Halbwertszeit von 1,25 Mil-

by using more or less restrictive conditions to find the relevant words:

```
Select the words strictly contained in rectangle
------------------------------------------------
Die Altersübereinstimmung deutete darauf hin,
engen, nur 50 Millionen Jahre großen
Gesteinshagel auf den Mond traf und dabei
hinterließ – einige größer als Frankreich.
es sich um eine letzte, infernalische Welle
Geburt des Sonnensystems. Daher tauften die
das Ereignis »lunare Katastrophe«. Später
die Bezeichnung Großes Bombardement durch.
```

Or, more forgiving, respectively:

```
Select the words intersecting the rectangle
-------------------------------------------
Die Altersübereinstimmung deutete darauf hin, dass
einem engen, nur 50 Millionen Jahre großen Zeitfenster
ein Gesteinshagel auf den Mond traf und dabei unzählige
Krater hinterließ – einige größer als Frankreich. Offenbar
handelte es sich um eine letzte, infernalische Welle nach
der Geburt des Sonnensystems. Daher tauften die Caltech-
Forscher das Ereignis »lunare Katastrophe«. Später setzte
sich die Bezeichnung Großes Bombardement durch.
```

The latter output also includes words *intersecting* the rectangle.

What if your **rectangle spans across more than one page**? Follow this recipe:

- Create a common list of all words of all pages which your rectangle intersects.
- When adding word items to this common list, increase their **y-coordinates** by the accumulated height of all previous pages.

How to Extract Text in Natural Reading Order

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

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

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

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

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

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

As a starting point take the above mentioned script and then use the full page rectangle.

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

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

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

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

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

    Notes:
        Method 'getTextWords()' does not try to recover words, if their single
        letters do not appear in correct lexical order. This function steps in
        here and creates a new list of recovered words.
    Args:
        words: list of words as created by 'getTextWords()'
        rect: rectangle to consider (usually the full page)
    Returns:
        List of recovered words. Same format as 'getTextWords', but left out
        block, line and word number - a list of items of the following format:
        [x0, y0, x1, y1, "word"]
    """
    # build my sublist of words contained in given rectangle
    mywords = [w for w in words if fitz.Rect(w[:4]) in rect]

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

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

    words_out = []  # we will return this

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

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

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

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

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

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

    return words_out

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

    Notes:
        Can be adjusted / extended in obvious ways, e.g. using regular
        expressions, or being case insensitive, or only looking for complete
        words, etc.
    Args:
        text: string to be searched for
        words: list of items in format delivered by 'getTextWords()'.
    Returns:
        List of rectangles, one for each found locations.
    """
    rect_list = []
    for w in words:
        if text in w[4]:
            rect_list.append(fitz.Rect(w[:4]))

    return rect_list
```

How to Extract Tables from Documents

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

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

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

How to Search for and Mark Text

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

This method has advantages and drawbacks. Pros are

- the search string can contain blanks and wrap across lines
- upper or lower cases are treated equal
- return may also be a list of Quad objects to precisely locate text that is **not parallel** to either axis.

Disadvantages:

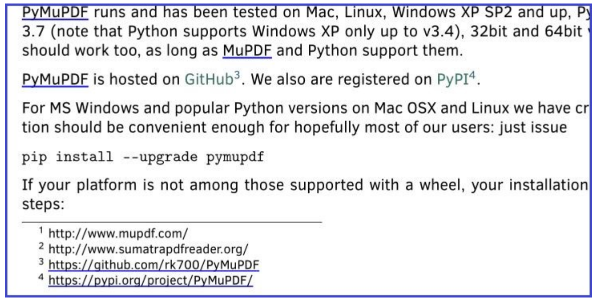
- you cannot determine the number of found items beforehand: if *hit_max* items are returned you do not know whether you have missed any.

But you have other options:

```python
import sys
import fitz

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

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

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

new_doc = False                        # indicator if anything found at all

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

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

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

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



PyMuPDF runs and has been tested on Mac, Linux, Windows XP SP2 and up, 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:

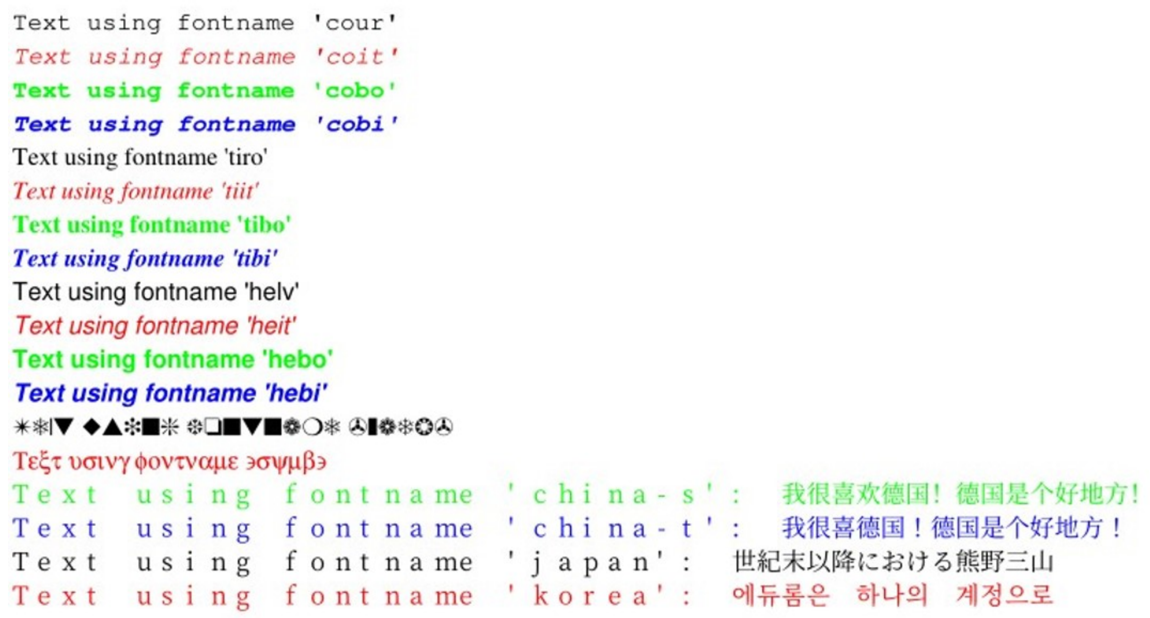
[1] http://www.mupdf.com/
[2] http://www.sumatrapdfreader.org/
[3] https://github.com/rk700/PyMuPDF
[4] https://pypi.org/project/PyMuPDF/

How to Analyze Font Characteristics

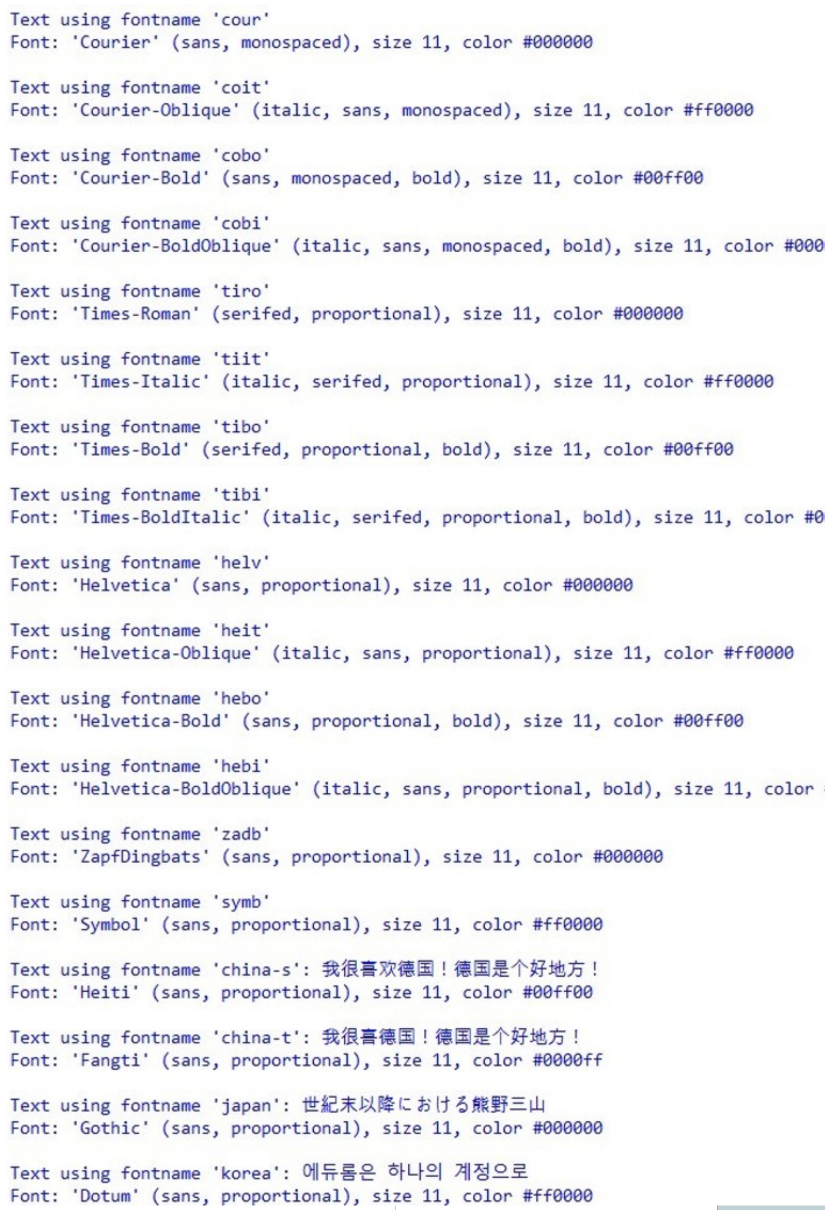
To analyze the characteristics of text in a PDF use this elementary script as a starting point:

```
import fitz


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


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

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

Here is the PDF page and the script output:



Text using fontname 'cour'
Text using fontname 'coit'
Text using fontname 'cobo'
Text using fontname 'cobi'
Text using fontname 'tiro'
Text using fontname 'tiit'
Text using fontname 'tibo'
Text using fontname 'tibi'
Text using fontname 'helv'
Text using fontname 'heit'
Text using fontname 'hebo'
Text using fontname 'hebi'
✳✻❙▼ ◆▲✽■✻ ❆❑■▼■❁❍✻ ✇❙❁✽✪✇
Τεξτ υσινγ φοντναμε ∍συμβ∍
T e x t   u s i n g   f o n t n a m e   ' c h i n a - s ' :   我很喜欢德国! 德国是个好地方!
T e x t   u s i n g   f o n t n a m e   ' c h i n a - t ' :   我很喜德国！德国是个好地方！
T e x t   u s i n g   f o n t n a m e   ' j a p a n ' :   世紀末以降における熊野三山
T e x t   u s i n g   f o n t n a m e   ' k o r e a ' :   에듀롬은 하나의 계정으로



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

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

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

Text using fontname 'cobi'
Font: 'Courier-BoldOblique' (italic, sans, monospaced, bold), size 11, color #000

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 #0

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

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
```

How to Insert Text

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

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

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

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

Note

Both text insertion methods automatically install the font as necessary.

How to Write Text Lines

Output some text lines on a page:

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

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

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

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

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

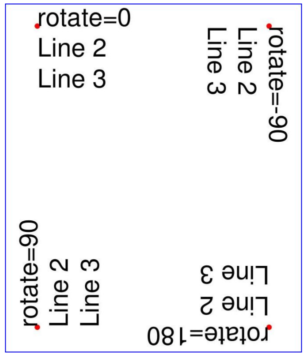
Line **width is ignored**. The surplus part of a line will simply be invisible.

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

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

```
import fitz
doc = fitz.open()
page = doc.newPage()
# the text strings, each having 3 lines
text1 = "rotate=0\nLine 2\nLine 3"
text2 = "rotate=90\nLine 2\nLine 3"
text3 = "rotate=-90\nLine 2\nLine 3"
text4 = "rotate=180\nLine 2\nLine 3"
red = (1, 0, 0) # the color for the red dots
# the insertion points, each with a 25 pix distance from the corners
p1 = fitz.Point(25, 25)
p2 = fitz.Point(page.rect.width - 25, 25)
p3 = fitz.Point(25, page.rect.height - 25)
p4 = fitz.Point(page.rect.width - 25, page.rect.height - 25)
# create a Shape to draw on
shape = page.newShape()

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

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

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

This is the result:

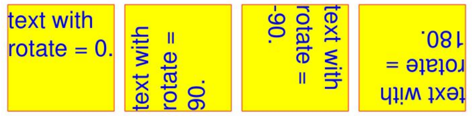


How to Fill a Text Box

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

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

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



How to Use Non-Standard Encoding

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

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

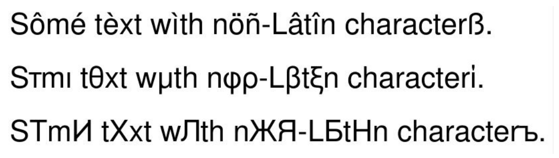
The valid encoding values are TEXT_ENCODING_LATIN (0), TEXT_ENCODING_GREEK (1), and TEXT_ENCODING_CYRILLIC (2, Russian) with Latin being the default. Encoding can be specified by all relevant font and text insertion methods.

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

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

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

The result:



Sômé tèxt wìth nöñ-Lâtîn characterß.

Sтmı tθxt wμth nφρ-Lβtξn characterΐ.

STmИ tXxt wЛth nЖЯ-LБtHn characterъ.

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

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

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

Annotations

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

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

How to Add and Modify Annotations

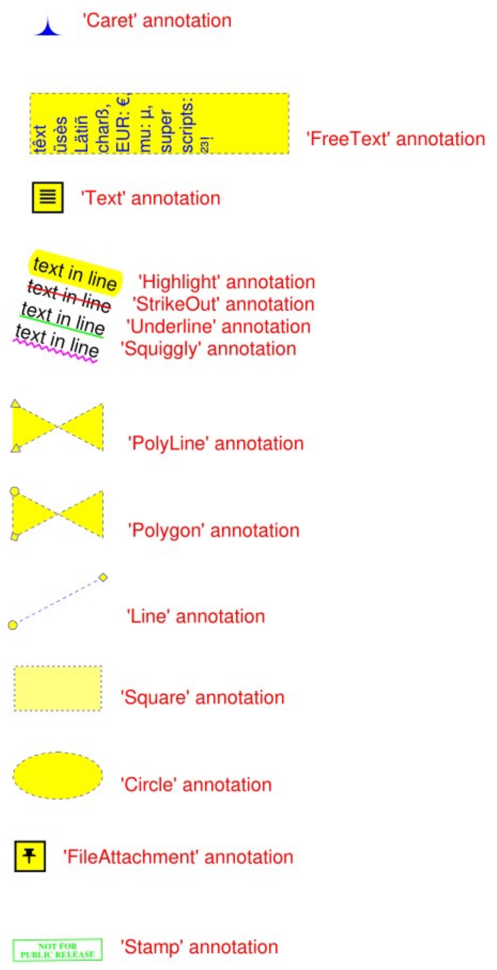
In PyMuPDF, new annotations are added via Page methods. To keep code duplication effort small, we only offer a minimal set of options here. For example, to add a 'Circle' annotation, only the containing rectangle can be specified. The result is a circle (or ellipsis) with white interior, black border and a line width of 1, exactly fitting into the rectangle. To adjust the annot's appearance, Annot methods must then be used. After having made all required changes, the annot's `Annot.update()` methods must be invoked to finalize all your changes.

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

```
# -*- coding: utf-8 -*-
from __future__ import print_function
import sys

import fitz

print(fitz.__doc__)
if fitz.VersionBind.split(".") < ["1", "16", "0"]:
    sys.exit("PyMuPDF v1.16.0+ is needed.")
"""
-------------------------------------------------------------------------------
Demo script showing how annotations can be added to a PDF using PyMuPDF.

It contains the following annotation types:
Text ("sticky note"), FreeText, text markers (underline, strike-out,
highlight), Circle, Square, Line, PolyLine, Polygon, FileAttachment and Stamp.


Dependencies
------------
PyMuPDF v1.16.0
-------------------------------------------------------------------------------
"""
text = "text in line\ntext in line\ntext in line\ntext in line"
red = (1, 0, 0)
blue = (0, 0, 1)
gold = (1, 1, 0)
green = (0, 1, 0)

displ = fitz.Rect(0, 50, 0, 50)
r = fitz.Rect(72, 100, 220, 135)
t1 = u"têxt üsès Lätiñ charß,\nEUR: €, mu: µ, super scripts: ²³!"


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


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

annot = page.addCaretAnnot(r.tl)  # 'Caret'
print_descr(annot.rect, annot)

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

print_descr(annot.rect, annot)
r = annot.rect + displ

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

#-------------------------------------------------------------------------------
# prepare insertion of 4 text highlight annotations
#-------------------------------------------------------------------------------
```

```
pos = annot.rect.tl + displ.tl
# 1. insert 4 rotated text lines
page.insertText(pos, text, fontsize=11, morph=(pos, fitz.Matrix(-15)))
# 2. search text to get the quads
rl = page.searchFor("text in line", quads=True)
r0 = rl[0]  # these are the 4 quads
r1 = rl[1]
r2 = rl[2]
r3 = rl[3]
annot = page.addHighlightAnnot(r0)
# need to convert quad to rect for descriptive text ...
print_descr(r0.rect, annot)

annot = page.addStrikeoutAnnot(r1)
print_descr(r1.rect, annot)

annot = page.addUnderlineAnnot(r2)
print_descr(r2.rect, annot)

annot = page.addSquigglyAnnot(r3)
print_descr(r3.rect, annot)
# end of text highlight annot code
#-------------------------------------------------------------------------------

r = r3.rect + displ
annot = page.addPolylineAnnot([r.bl, r.tr, r.br, r.tl])  # 'Polyline'
annot.setBorder(width=0.3, dashes=[2])
annot.setColors(stroke=blue, fill=gold)
annot.setLineEnds(fitz.PDF_ANNOT_LE_CLOSED_ARROW,
                  fitz.PDF_ANNOT_LE_R_CLOSED_ARROW)
annot.update()
print_descr(annot.rect, annot)

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

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

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

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

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

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


doc.save("new-annots.pdf")
```

This script should lead to the following output:

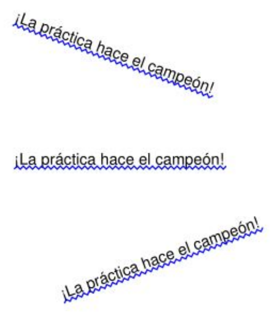


How to Mark Text

This script searches for text and marks it:

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

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

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

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

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

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

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

The result looks like this:


¡La práctica hace el campeón!
¡La práctica hace el campeón!
¡La práctica hace el campeón!

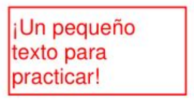
How to Use FreeText

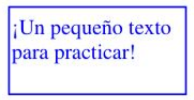
This script shows a couple of ways to deal with 'FreeText' annotations:

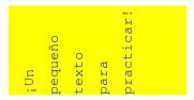
```
# -*- coding: utf-8 -*-
import fitz

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

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

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

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

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

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

The result looks like this:







Using Buttons and JavaScript

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

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

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

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

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

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

Use PyMuPDF v1.16.12 to create a push button that executes a Javascript
containing the desired code. This is what this program does.
Then open the resulting file with Adobe reader (!).
After clicking on the button, all Freetext annotations will be bold, and the
file can be saved.
If desired, the button can be removed again, using free tools like PyMuPDF or
PDF XChange editor.

Note / Caution:
---------------
The JavaScript will **only** work if the file is opened with Adobe Acrobat reader!
When using other PDF viewers, the reaction is unforeseeable.
"""
import sys

import fitz

# this JavaScript will execute when the button is clicked:
jscript = """
var annt = this.getAnnots();
annt.forEach(function (item, index) {
    try {
        var span = item.richContents;
        span.forEach(function (it, dx) {
            it.fontWeight = 800;
        })
        item.richContents = span;
    } catch (err) {}
});
app.alert('Done');
"""
i_fn = sys.argv[1]  # input file name
o_fn = "bold-" + i_fn  # output filename
doc = fitz.open(i_fn)  # open input
page = doc[0]  # get desired page

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

widget = fitz.Widget()  # create widget

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

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

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

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

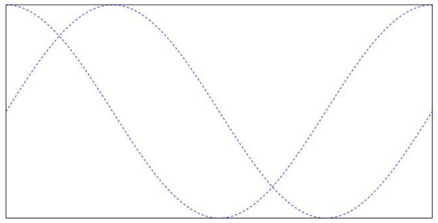
How to Use Ink Annotations

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

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

```python
import math
import fitz

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

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

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

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

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

This is the result:



Drawing and Graphics

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

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

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

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

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

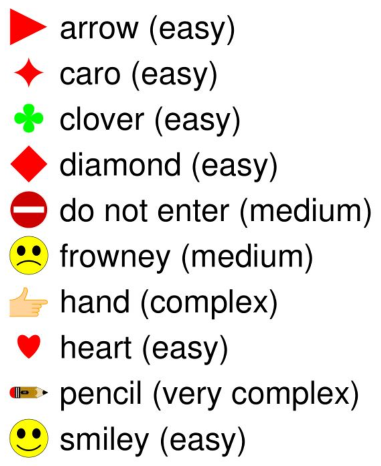
In fact, `Shape.finish()` *defines* a group of preceding draw operations to form one – potentially rather complex – graphics object. PyMuPDF provides several predefined graphics in shapes_and_symbols.py which demonstrate how this works.

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

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

@author: Jorj
@license: GNU GPL 3.0+

Create a list of available symbols defined in shapes_and_symbols.py

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

"""

import fitz
import shapes_and_symbols as sas

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

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

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

doc = fitz.open()  # create empty PDF
page = doc.newPage()  # create an empty page
```

```
shape = page.newShape()  # start a Shape (canvas)

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

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

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

This is the script’s outcome:



Multiprocessing

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

No Python threading support.

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

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

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

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

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

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

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

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


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

    Notes:
        The PyMuPDF document cannot be part of the argument, because that
        cannot be pickled. So we are being passed in just its filename.
        This is no performance issue, because we are a separate process and
        need to open the document anyway.
        Any page-specific function can be processed here - rendering is just
        an example - text extraction might be another.
        The work must however be self-contained: no inter-process communication
        or synchronization is possible with this design.
        Care must also be taken with which parameters are contained in the
        argument, because it will be passed in via pickling by the Pool class.
        So any large objects will increase the overall duration.
    Args:
        vector: a list containing required parameters.
    """
    # recreate the arguments
    idx = vector[0]  # this is the segment number we have to process
    cpu = vector[1]  # number of CPUs
    filename = vector[2]  # document filename
    mat = vector[3]  # the matrix for rendering
    doc = fitz.open(filename)  # open the document
    num_pages = len(doc)  # get number of pages

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

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


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

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

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

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

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

```
"""
Created on 2019-05-01

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

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

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


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

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

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

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

        self.btnStop = QtWidgets.QPushButton("Stop", self)
        self.btnStop.clicked.connect(self.stopPlay)
        hbox.addWidget(self.btnStop)
        self.label = QtWidgets.QLabel("0/0", self)
        self.label.setFont(QtGui.QFont("Verdana", 20))
        hbox.addWidget(self.label)

        vbox.addLayout(hbox)

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

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

    def openDoc(self):
        path, _ = QtWidgets.QFileDialog.getOpenFileName(
            self,
            "Open Document",
            self.lastDir,
            "All Supported Files (*.pdf;*.epub;*.xps;*.oxps;*.cbz;*.fb2);;PDF Files (*.pdf);;EPUB Files (*.epub);;XPS Files (*.xps);;Oper
            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.pageCount = 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.pageCount - 1:
            self.queNum.put(self.curPageNum + 1)
        else:
            self.timerSend.stop()

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

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

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


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


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

General

How to Open with a Wrong File Extension

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

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

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

Note

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

How to Embed or Attach Files

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

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

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

There exist several example scripts: embedded-list.py, new-annots.py.

Also look at the sections above and at chapter Appendix 3: Considerations on Embedded Files.

How to Delete and Re-Arrange Pages

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

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

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

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

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

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

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

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

For more information also have a look at this Wiki article.

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

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

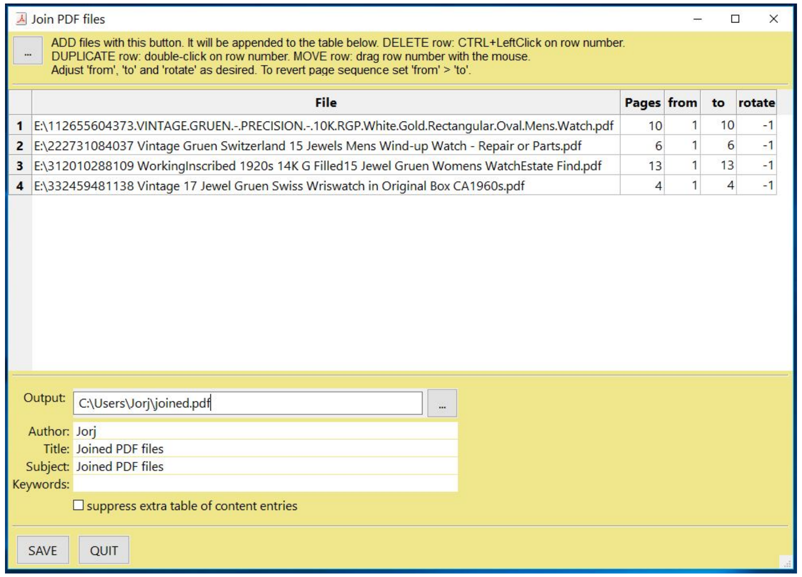
This snippet duplicates the PDF with itself so that it will contain the pages *0, 1, …, n, 0, 1, …, n* **(extremely fast and without noticeably increasing the file size!)**:

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

How to Join PDFs

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

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



ADD files with this button. It will be appended to the table below. DELETE row: CTRL+LeftClick on row number.
DUPLICATE row: double-click on row number. MOVE row: drag row number with the mouse.
Adjust 'from', 'to' and 'rotate' as desired. To revert page sequence set 'from' > 'to'.

	File	Pages	from	to	rotate
1	E:\112655604373.VINTAGE.GRUEN.-.PRECISION.-.10K.RGP.White.Gold.Rectangular.Oval.Mens.Watch.pdf	10	1	10	-1
2	E:\222731084037 Vintage Gruen Switzerland 15 Jewels Mens Wind-up Watch - Repair or Parts.pdf	6	1	6	-1
3	E:\312010288109 WorkingInscribed 1920s 14K G Filled15 Jewel Gruen Womens WatchEstate Find.pdf	13	1	13	-1
4	E:\332459481138 Vintage 17 Jewel Gruen Swiss Wriswatch in Original Box CA1960s.pdf	4	1	4	-1

Output: C:\Users\Jorj\joined.pdf
Author: Jorj
Title: Joined PDF files
Subject: Joined PDF files
Keywords:
☐ suppress extra table of content entries

How to Add Pages

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

newPage

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

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

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

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

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

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

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

Note

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

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

insertPage

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

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

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

How To Dynamically Clean Up Corrupt PDFs

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

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

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

#----------------------------------------
# 'Tolerant' PDF reader
#----------------------------------------
def reader(fname, password = None):
    idata = open(fname, "rb").read()  # read the PDF into memory and
    ibuffer = BytesIO(idata)  # convert to stream
    if password is None:
        try:
            return PdfReader(ibuffer)  # if this works: fine!
        except:
            pass

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

How to Split Single Pages

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

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

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

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

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

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

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

for spage in src:  # for each page in input
    r = spage.rect  # input page rectangle
    d = fitz.Rect(spage.CropBoxPosition,  # CropBox displacement if not
                  spage.CropBoxPosition)  # starting at (0, 0)
    #--------------------------------------------------------------------------
    # example: cut input page into 2 x 2 parts
    #--------------------------------------------------------------------------
    r1 = r * 0.5  # top left rect
    r2 = r1 + (r1.width, 0, r1.width, 0)  # top right rect
    r3 = r1 + (0, r1.height, 0, r1.height)  # bottom left rect
    r4 = fitz.Rect(r1.br, r.br)  # bottom right rect
    rect_list = [r1, r2, r3, r4]  # put them in a list

    for rx in rect_list:  # run thru rect list
        rx += d  # add the CropBox displacement
        page = doc.newPage(-1,  # new output page with rx dimensions
                           width = rx.width,
                           height = rx.height)
        page.showPDFpage(
                page.rect,  # fill all new page with the image
                src,  # input document
                spage.number,  # input page number
                clip = rx,  # which part to use of input page
            )

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

This shows what happens to an input page:

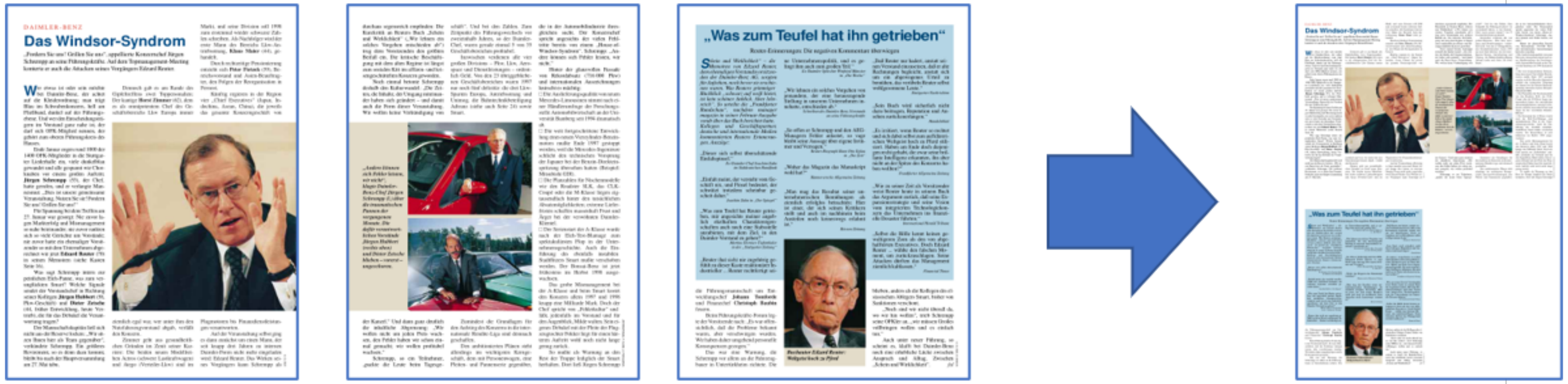
How to Combine Single Pages

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

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

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

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

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

(2) Easily adapt the example to combine just 2 pages (like for a booklet) or
```

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

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

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

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

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

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

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

Example effect:



How to Convert Any Document to PDF

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

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

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

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

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

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

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

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

doc = fitz.open(fn)

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

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

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

# now process the links
link_cnti = 0
link_skip = 0
for pinput in doc:                # iterate through input pages
    links = pinput.getLinks()     # 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"] == fitz.LINK_NAMED:    # we do not handle named links
            print("named link page", pinput.number, l)
            link_skip += 1        # count them
            continue
        pout.insertLink(l)        # simply output the others

# save the conversion result
pdf.save(fn + ".pdf", garbage=4, deflate=True)
# say how many named links we skipped
```

```
if link_cnti > 0:
    print("Skipped %i named links of a total of %i in input." % (link_skip, link_cnti))
```

How to Deal with Messages Issued by MuPDF

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

We always prefix these messages with an identifying string *"mupdf:"*.

MuPDF warnings continue to be stored in an internal buffer and can be viewed using `Tools.mupdf_warnings()`. Please note that MuPDF errors may or may not lead to Python exceptions. In other words, you may see error messages from which MuPDF can recover and continue processing.

Example output for a **recoverable error**. We are opening a damaged PDF, but MuPDF is able to repair it and gives us a few information on what happened. Then we illustrate how to find out whether the document can later be saved incrementally:

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

Example output for an **unrecoverable error**:

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

How to Deal with PDF Encryption

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

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

- set permission details

Note

A PDF document may have two different passwords:

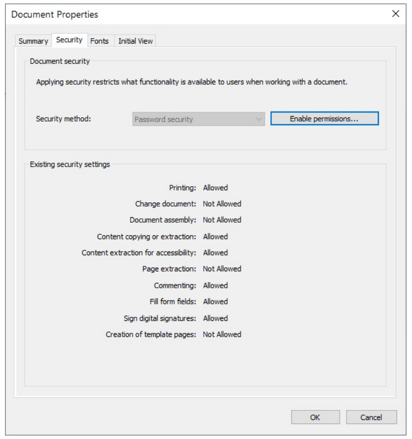
- The **owner password** provides full access rights, including changing passwords, encryption method, or permission detail.
- The **user password** provides access to document content according to the established permission details. If present, opening the PDF in a viewer will require providing it.

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

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

```
import fitz

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

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



Document security

Applying security restricts what functionality is available to users when working with a document.

Security method: Password security Enable permissions...

Existing security settings

Printing:	Allowed
Change document:	Not Allowed
Document assembly:	Not Allowed
Content copying or extraction:	Allowed
Content extraction for accessibility:	Allowed
Page extraction:	Not Allowed
Commenting:	Allowed
Fill form fields:	Allowed
Sign digital signatures:	Allowed
Creation of template pages:	Not Allowed

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

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

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

Common Issues and their Solutions

Changing Annotations: Unexpected Behaviour

Problem

There are two scenarios:

1. Updating an annotation, which has been created by some other software, via a PyMuPDF script.
2. Creating an annotation with PyMuPDF and later changing it using some other PDF application.

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.

Cause

Annotation maintenance is handled differently by each PDF maintenance application (if it is supported at all). For any given PDF application, some annotation types may not be supported at all or only partly, or some details may be handled in a different way than with another application.

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 fully supports the PDF text markers, 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.

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.setRect()` (location changes)
    - `Annot.setFlags()` (annotation behaviour)
    - `Annot.setInfo()` (meta information, except changes to *content*)
    - `Annot.fileUpd()` (file attachment changes)

Misplaced Item Insertions on PDF Pages

Problem

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

Cause

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

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

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

So the PDF creator did this:

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

where they should have done this:

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

Note

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

Solutions

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

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

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

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

Note

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

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

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

Note

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

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

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 lateron assessed as being part of the normal interface. This has happened in v1.14.0 for the class Tools – you now find it as an item in the Classes chapter.

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

How to Iterate through the `xref` Table

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

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

This produces the following output:

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

object 2 (stream: False)
<<
```

```
    /Type /Catalog
    /Pages 3 0 R
>>

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

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

A PDF object definition is an ordinary ASCII string.

How to Handle Object Streams

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

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

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

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

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

How to Handle Page Contents

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

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

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

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

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

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

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

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

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

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

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

How to Access the PDF Catalog

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

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

Note

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

How to Access the PDF File Trailer

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

Key	Type	Value
Size	int	Number of entries in the cross-reference table + 1.
Prev	int	Offset to previous 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. 109.

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

```
>>> import fitz
>>> doc=fitz.open("PyMuPDF.pdf")
>>> trailer=doc._getTrailerString()
>>> print(trailer)
<</Size 5535/Info 5275 0 R/Root 5274 0 R/ID[(\340\273fE\225^l\226\232O|\003\201\325g\245)(}#1,\317\205\000\371\251wO6\352Oa\021)]>>
>>>
```

How to Access XML Metadata

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

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

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

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

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

Tutorial

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

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

Importing the Bindings

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

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

Opening a Document

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

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

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

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

Table of Contents

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

Some Document Methods and Attributes

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

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

Key	Value
creator	creating application
subject	subject

Note

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

Note

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

Working with Outlines

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

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

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

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

Note

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

Working with Pages

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

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

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

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

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

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

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

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

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

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

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 taget that is encoded in that hot area. Here is how to get all links:

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

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

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

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

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

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

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

Rendering a Page

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

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

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

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

Note

You can also create a **vector** image of a page by using `Page.getSVGimage()`. Refer to this Wiki for details.

Saving the Page Image in a File

We can simply store the image in a PNG file:

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

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

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

PyQt4, PyQt5, PySide

Please also see section 3.16 of the Pillow documentation:

```
from PIL import Image, ImageQt

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

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

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

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

Extracting Text and Images

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

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

Use one of the following strings for *opt* to obtain different 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()` resp. `TextPage.extractJSON()` for details of its structure.
- *"rawdict"*: a super-set of `TextPage.extractDICT()`. It additionally provides character detail information like XML. See `TextPage.extractRAWDICT()` for details of its structure.
- *"xhtml"*: text information level as the TEXT version but includes images. Can also be displayed by internet browsers.
- *"xml"*: contains no images, but full position and font information down to each single text character. Use an XML module to interpret.

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

Searching for Text

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

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

This delivers a list of up to 16 rectangles (see Rect), each of which surrounds one occurrence of the string "mupdf" (case insensitive). You could use this information to e.g. highlight those areas (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.

PDF Maintenance

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

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

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

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

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

Modifying, Creating, Re-arranging and Deleting Pages

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

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

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

`Document.select()` shrinks a PDF down to selected pages. Parameter is a sequence [3] of the page numbers that you want to keep. These integers must all be in range *0 <= i < pageCount*. When executed, all pages **missing** in this list will be deleted. Remaining pages will occur **in the sequence and as many times (!) as you specify them**.

So you can easily create new PDFs with

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

… whatever you can think of.

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

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

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

Joining and Splitting PDF Documents

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

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

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

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

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

Embedding Data

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

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

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

Save Option	mutool	Effect
garbage=2	gg	in addition to 1, compact `xref` tables
garbage=3	ggg	in addition to 2, merge duplicate objects
garbage=4	gggg	in addition to 3, skip duplicate streams
clean=1	cs	clean and sanitize content streams
deflate=1	z	deflate uncompressed streams
ascii=1	a	convert binary data to ASCII format
linear=1	l	create a linearized version
expand=1	i	decompress images
expand=2	f	decompress fonts
expand=255	d	decompress all

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

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.

Further Reading

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

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

Footnotes

[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.

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

[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.

Tutorial

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

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

Importing the Bindings

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

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

Opening a Document

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

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

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

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

Table of Contents

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

Some Document Methods and Attributes

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

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

Previous topic

Installation

Next topic

Collection of Recipes

Quick search

Go

Key	Value
creator	creating application
subject	subject

Note

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

Note

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

Working with Outlines

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

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

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

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

Note

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

Working with Pages

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

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

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

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

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

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

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

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

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

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

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 taget that is encoded in that hot area. Here is how to get all links:

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

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

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

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

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

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

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

Rendering a Page

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

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

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

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

Note

You can also create a **vector** image of a page by using `Page.getSVGimage()`. Refer to this Wiki for details.

Saving the Page Image in a File

We can simply store the image in a PNG file:

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

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

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

PyQt4, PyQt5, PySide

Please also see section 3.16 of the Pillow documentation:

```
from PIL import Image, ImageQt

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

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

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

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

Extracting Text and Images

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

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

Use one of the following strings for *opt* to obtain different 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()` resp. `TextPage.extractJSON()` for details of its structure.
- *"rawdict"*: a super-set of `TextPage.extractDICT()`. It additionally provides character detail information like XML. See `TextPage.extractRAWDICT()` for details of its structure.
- *"xhtml"*: text information level as the TEXT version but includes images. Can also be displayed by internet browsers.
- *"xml"*: contains no images, but full position and font information down to each single text character. Use an XML module to interpret.

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

Searching for Text

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

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

This delivers a list of up to 16 rectangles (see Rect), each of which surrounds one occurrence of the string "mupdf" (case insensitive). You could use this information to e.g. highlight those areas (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.

PDF Maintenance

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

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

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

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

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

Modifying, Creating, Re-arranging and Deleting Pages

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

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

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

`Document.select()` shrinks a PDF down to selected pages. Parameter is a sequence [3] of the page numbers that you want to keep. These integers must all be in range *0 <= i < pageCount*. When executed, all pages **missing** in this list will be deleted. Remaining pages will occur **in the sequence and as many times (!) as you specify them**.

So you can easily create new PDFs with

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

… whatever you can think of.

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

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

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

Joining and Splitting PDF Documents

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

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

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

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

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

Embedding Data

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

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

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

Save Option	mutool	Effect
garbage=2	gg	in addition to 1, compact `xref` tables
garbage=3	ggg	in addition to 2, merge duplicate objects
garbage=4	gggg	in addition to 3, skip duplicate streams
clean=1	cs	clean and sanitize content streams
deflate=1	z	deflate uncompressed streams
ascii=1	a	convert binary data to ASCII format
linear=1	l	create a linearized version
expand=1	i	decompress images
expand=2	f	decompress fonts
expand=255	d	decompress all

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

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.

Further Reading

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

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

Footnotes

[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.

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

[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.