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
```

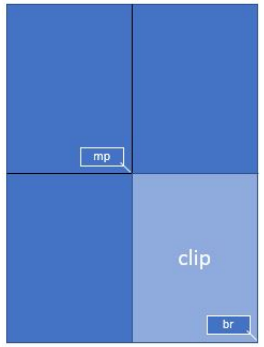
How to Create Partial Pixmaps (Clips)

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:

Previous topic

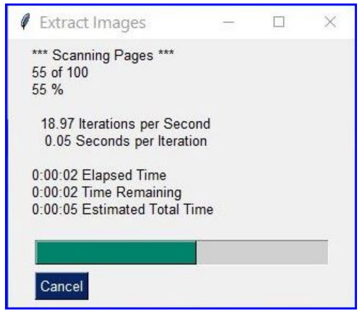
Tutorial

Next topic

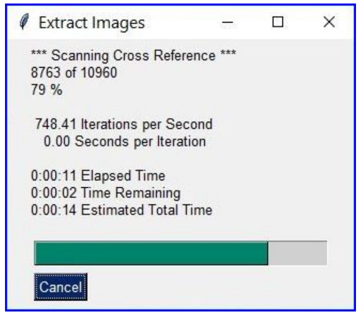
Using *fitz* as a Module

Quick search

Go


```
*** 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

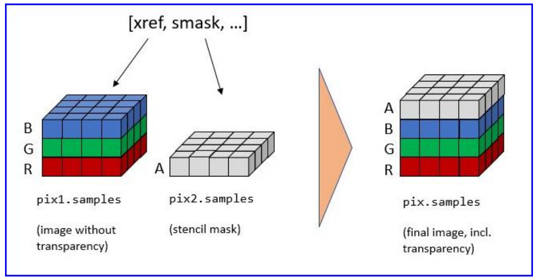
Some images in PDFs are accompanied by **stencil masks**. In their simplest form stencil masks represent alpha (transparency) bytes stored as seperate images. In order to reconstruct the original of an image, which has a stencil mask, it must be "enriched" with transparency bytes taken from its stencil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

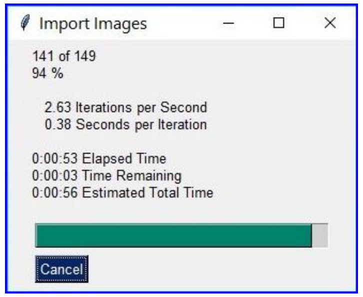
Method 1: Inserting Images as Pages

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.

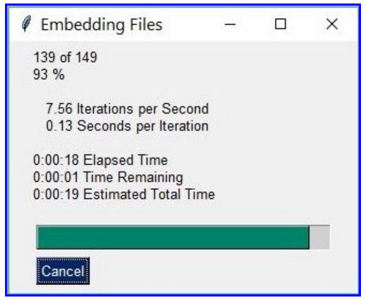
Note

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")
```



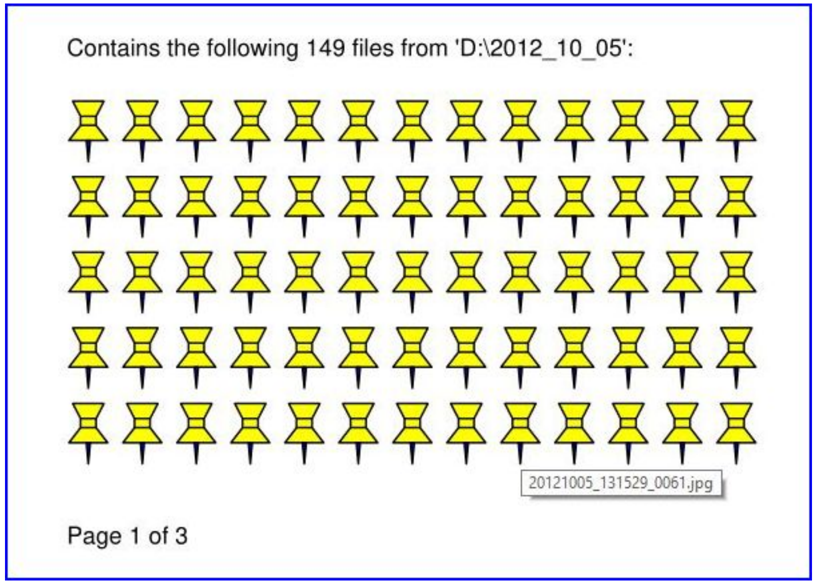
```
139 of 149
93 %

  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.


Contains the following 149 files from 'D:\2012_10_05':

Page 1 of 3

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:

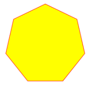
```
if str is bytes:  # this is Python 2!
    import Tkinter as tk
else:  # Python 3 or later!
    import tkinter as tk
pix = fitz.Pixmap("input.png")  # may have an alpha channel
if pix.alpha:  # we have an alpha channel!
    pix = fitz.Pixmap(pix, 0)  # remove it
tkimg = tk.PhotoImage(data=pix.getImageData("ppm"))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

How to Use Pixmaps: Glueing Images

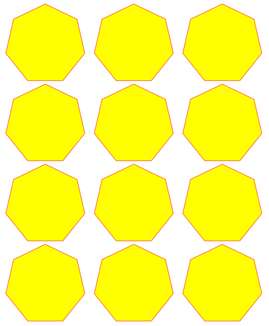
This shows how pixmaps can be used for purely graphical, non-document purposes. The script reads an image file and creates a new image which consist of 3 * 4 tiles of the original:

```
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:

[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.