pdfminer.six

Release 20220506

Yusuke Shinyama, Philippe Guglielmetti & Pieter Marsman

May 07, 2022

CONTENTS

We fathom PDF.

Pdfminer.six is a python package for extracting information from PDF documents.

Check out the source on github.

CHAPTER ONE

CONTENT

This documentation is organized into four sections (according to the Diátaxis documentation framework). The *Tutorials* section helps you setup and use pdfminer.six for the first time. Read this section if this is your first time working with pdfminer.six. The *How-to guides* offers specific recipies for solving common problems. Take a look at the *Topics* if you want more background information on how pdfminer.six works internally. The *API Reference* provides detailed api documentation for all the common classes and functions in pdfminer.six.

1.1 Tutorials

Tutorials help you get started with specific parts of pdfminer.six.

1.1.1 Install pdfminer.six as a Python package

To use pdfminer.six for the first time, you need to install the Python package in your Python environment.

This tutorial requires you to have a system with a working Python and pip installation. If you don't have one and don't know how to install it, take a look at The Hitchhiker's Guide to Python!.

Install using pip

Run the following command on the commandline to install pdfminer.six as a Python package:

```
pip install pdfminer.six
```

Test pdfminer.six installation

You can test the pdfminer.six installation by importing it in Python.

Open an interactive Python session from the commandline import pdfminer .six:

```
>>> import pdfminer
>>> print(pdfminer.__version__)
'<installed version>'
```

Now you can use pdfminer.six as a Python package. But pdfminer.six also comes with a couple of useful commandline tools. To test if these tools are correctly installed, run the following on your commandline:

```
$ pdf2txt.py --version
pdfminer.six <installed version>
```

1.1.2 Extract text from a PDF using the commandline

pdfminer.six has several tools that can be used from the command line. The command-line tools are aimed at users that occasionally want to extract text from a pdf.

Take a look at the high-level or composable interface if you want to use pdfminer.six programmatically.

Examples

pdf2txt.py

```
$ python tools/pdf2txt.py example.pdf
all the text from the pdf appears on the command line
```

The *pdf2txt.py* tool extracts all the text from a PDF. It uses layout analysis with sensible defaults to order and group the text in a sensible way.

dumppdf.py

```
$ python tools/dumppdf.py -a example.pdf
<pdf><object id="1">
...
</object>
...
</pdf>
```

The *dumppdf.py* tool can be used to extract the internal structure from a PDF. This tool is primarily for debugging purposes, but that can be useful to anybody working with PDF's.

1.1.3 Extract text from a PDF using Python

The high-level API can be used to do common tasks.

The most simple way to extract text from a PDF is to use *extract_text*:

```
>>> from pdfminer.high_level import extract_text
>>> text = extract_text('samples/simple1.pdf')
>>> print(repr(text))
'Hello \n\nWorld\n\nHello \n\nWorld\n\nH e l l o  \n\nW o r l d\n\nH e l l o  \n\nW o␣
↪r l d\n\n\x0c'
>>> print(text)
...
Hello

World

Hello

World

H e l l o

W o r l d
```

(continues on next page)

(continued from previous page)

```

H e l l o

W o r l d
```

To read text from a PDF and print it on the command line:

```
>>> from io import StringIO
>>> from pdfminer.high_level import extract_text_to_fp
>>> output_string = StringIO()
>>> with open('samples/simple1.pdf', 'rb') as fin:
...     extract_text_to_fp(fin, output_string)
>>> print(output_string.getvalue().strip())
Hello WorldHello WorldHello WorldHello World
```

Or to convert it to html and use layout analysis:

```
>>> from io import StringIO
>>> from pdfminer.high_level import extract_text_to_fp
>>> from pdfminer.layout import LAParams
>>> output_string = StringIO()
>>> with open('samples/simple1.pdf', 'rb') as fin:
...     extract_text_to_fp(fin, output_string, laparams=LAParams(),
...                        output_type='html', codec=None)
```

1.1.4 Extract text from a PDF using Python - part 2

The command line tools and the high-level API are just shortcuts for often used combinations of pdfminer.six components. You can use these components to modify pdfminer.six to your own needs.

For example, to extract the text from a PDF file and save it in a python variable:

```
from io import StringIO

from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from pdfminer.pdfdocument import PDFDocument
from pdfminer.pdfinterp import PDFResourceManager, PDFPageInterpreter
from pdfminer.pdfpage import PDFPage
from pdfminer.pdfparser import PDFParser

output_string = StringIO()
with open('samples/simple1.pdf', 'rb') as in_file:
    parser = PDFParser(in_file)
    doc = PDFDocument(parser)
    rsrcmgr = PDFResourceManager()
    device = TextConverter(rsrcmgr, output_string, laparams=LAParams())
    interpreter = PDFPageInterpreter(rsrcmgr, device)
    for page in PDFPage.create_pages(doc):
        interpreter.process_page(page)

print(output_string.getvalue())
```

1.1.5 Extract elements from a PDF using Python

The high level functions can be used to achieve common tasks. In this case, we can use api_extract_pages:

```
from pdfminer.high_level import extract_pages
for page_layout in extract_pages("test.pdf"):
    for element in page_layout:
        print(element)
```

Each `element` will be an `LTTextBox`, `LTFigure`, `LTLine`, `LTRect` or an `LTImage`. Some of these can be iterated further, for example iterating though an `LTTextBox` will give you an `LTTextLine`, and these in turn can be iterated through to get an `LTChar`. See the diagram here: *Layout analysis algorithm*.

Let's say we want to extract all of the text. We could do:

```
from pdfminer.high_level import extract_pages
from pdfminer.layout import LTTextContainer
for page_layout in extract_pages("test.pdf"):
    for element in page_layout:
        if isinstance(element, LTTextContainer):
            print(element.get_text())
```

Or, we could extract the fontname or size of each individual character:

```
from pdfminer.high_level import extract_pages
from pdfminer.layout import LTTextContainer, LTChar
for page_layout in extract_pages("test.pdf"):
    for element in page_layout:
        if isinstance(element, LTTextContainer):
            for text_line in element:
                for character in text_line:
                    if isinstance(character, LTChar):
                        print(character.fontname)
                        print(character.size)
```

1.2 How-to guides

How-to guides help you to solve specific problems with pdfminer.six.

1.2.1 How to extract images from a PDF

Before you start, make sure you have *installed pdfminer.six*. The second thing you need is a PDF with images. If you don't have one, you can download this research paper with images of cats and dogs and save it as *example.pdf*:

```
$ curl https://www.robots.ox.ac.uk/~vgg/publications/2012/parkhi12a/parkhi12a.pdf --
↪output example.pdf
```

Then run the *pdf2txt* command:

```
$ pdf2txt.py example.pdf --output-dir cats-and-dogs
```

This command extracts all the images from the PDF and saves them into the *cats-and-dogs* directory.

1.2.2 How to extract AcroForm interactive form fields from a PDF using PDFMiner

Before you start, make sure you have *installed pdfminer.six*.

The second thing you need is a PDF with AcroForms (as found in PDF files with fillable forms or multiple choices). There are some examples of these in the GitHub repository under *samples/acroform*.

Only AcroForm interactive forms are supported, XFA forms are not supported.

```python
from pdfminer.pdfparser import PDFParser
from pdfminer.pdfdocument import PDFDocument
from pdfminer.pdftypes import resolve1
from pdfminer.psparser import PSLiteral, PSKeyword
from pdfminer.utils import decode_text


data = {}


def decode_value(value):

    # decode PSLiteral, PSKeyword
    if isinstance(value, (PSLiteral, PSKeyword)):
        value = value.name

    # decode bytes
    if isinstance(value, bytes):
        value = decode_text(value)

    return value


with open(file_path, 'rb') as fp:
    parser = PDFParser(fp)

    doc = PDFDocument(parser)
    res = resolve1(doc.catalog)

    if 'AcroForm' not in res:
        raise ValueError("No AcroForm Found")

    fields = resolve1(doc.catalog['AcroForm'])['Fields']  # may need further resolving

    for f in fields:
        field = resolve1(f)
        name, values = field.get('T'), field.get('V')

        # decode name
        name = decode_text(name)

        # resolve indirect obj
        values = resolve1(values)

        # decode value(s)
        if isinstance(values, list):
            values = [decode_value(v) for v in values]
        else:
            values = decode_value(values)
```

(continues on next page)

(continued from previous page)

```
        data.update({name: values})

        print(name, values)
```

This code snippet will print all the fields name and value and save them in the “data” dictionary.

How it works:

- Initialize the parser and the PDFDocument objects

```
parser = PDFParser(fp)
doc = PDFDocument(parser)
```

- Get the catalog

(the catalog contains references to other objects defining the document structure, see section 7.7.2 of PDF 32000-1:2008 specs: https://www.adobe.com/devnet/pdf/pdf_reference.html)

```
res = resolve1(doc.catalog)
```

- Check if the catalog contains the AcroForm key and raise ValueError if not

(the PDF does not contain Acroform type of interactive forms if this key is missing in the catalog, see section 12.7.2 of PDF 32000-1:2008 specs)

```
if 'AcroForm' not in res:
    raise ValueError("No AcroForm Found")
```

- Get the field list resolving the entry in the catalog

```
fields = resolve1(doc.catalog['AcroForm'])['Fields']
for f in fields:
    field = resolve1(f)
```

- Get field name and field value(s)

```
name, values = field.get('T'), field.get('V')
```

- Decode field name.

```
name = decode_text(name)
```

- Resolve indirect field value objects

```
values = resolve1(value)
```

- Call the value(s) decoding method as needed

(a single field can hold multiple values, for example a combo box can hold more than one value at time)

```
if isinstance(values, list):
    values = [decode_value(v) for v in values]
else:
    values = decode_value(values)
```

(the decode_value method takes care of decoding the fields value returning a string)

- Decode PSLiteral and PSKeyword field values

```
if isinstance(value, (PSLiteral, PSKeyword)):
    value = value.name
```

- Decode bytes field values

```
if isinstance(value, bytes):
    value = utils.decode_text(value)
```

1.3 Topics

1.3.1 Converting a PDF file to text

Most PDF files look like they contain well structured text. But the reality is that a PDF file does not contain anything that resembles paragraphs, sentences or even words. When it comes to text, a PDF file is only aware of the characters and their placement.

This makes extracting meaningful pieces of text from PDF files difficult. The characters that compose a paragraph are no different from those that compose the table, the page footer or the description of a figure. Unlike other document formats, like a *.txt* file or a word document, the PDF format does not contain a stream of text.

A PDF document does consists of a collection of objects that together describe the appearance of one or more pages, possibly accompanied by additional interactive elements and higher-level application data. A PDF file contains the objects making up a PDF document along with associated structural information, all represented as a single self-contained sequence of bytes.[1]

Layout analysis algorithm

PDFMiner attempts to reconstruct some of those structures by using heuristics on the positioning of characters. This works well for sentences and paragraphs because meaningful groups of nearby characters can be made.

The layout analysis consists of three different stages: it groups characters into words and lines, then it groups lines into boxes and finally it groups textboxes hierarchically. These stages are discussed in the following sections. The resulting output of the layout analysis is an ordered hierarchy of layout objects on a PDF page.

The output of the layout analysis heavily depends on a couple of parameters. All these parameters are part of the *LAParams* class.

Grouping characters into words and lines

The first step in going from characters to text is to group characters in a meaningful way. Each character has an x-coordinate and a y-coordinate for its bottom-left corner and upper-right corner, i.e. its bounding box. Pdfminer.six uses these bounding boxes to decide which characters belong together.

Characters that are both horizontally and vertically close are grouped onto one line. How close they should be is determined by the *char_margin* (M in figure) and the *line_overlap* (not in figure) parameter. The horizontal *distance* between the bounding boxes of two characters should be smaller than the *char_margin* and the vertical *overlap* between the bounding boxes should be smaller than the *line_overlap*.

The values of *char_margin* and *line_overlap* are relative to the size of the bounding boxes of the characters. The *char_margin* is relative to the maximum width of either one of the bounding boxes, and the *line_overlap* is relative to the minimum height of either one of the bounding boxes.

[1] Adobe System Inc. (2007). *Pdf reference: Adobe portable document format, version 1.7.*

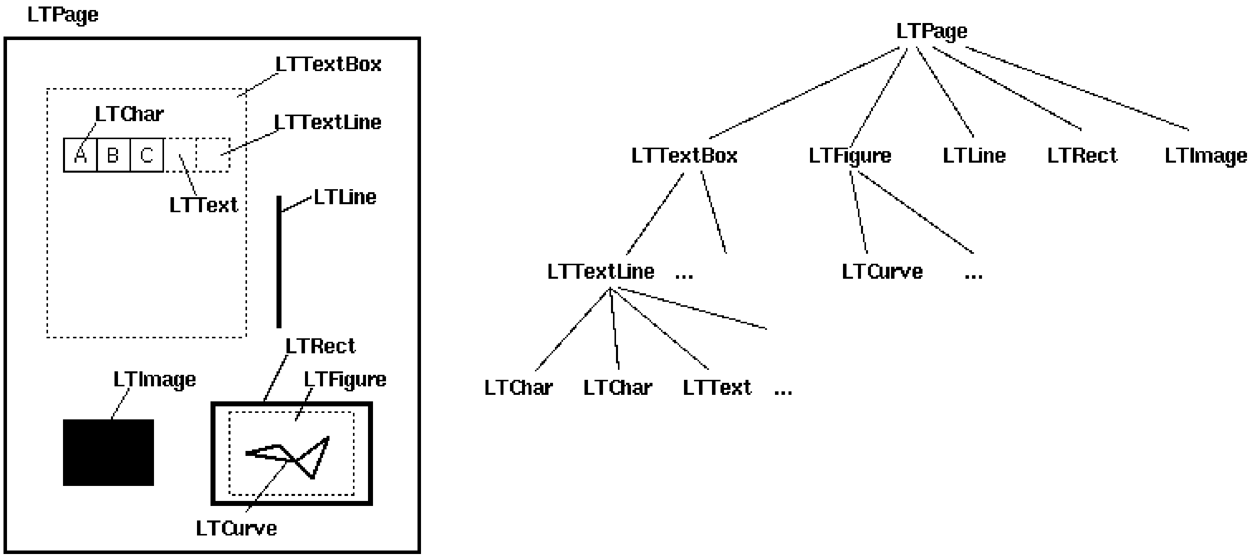

Fig. 1: The output of the layout analysis is a hierarchy of layout objects.

Spaces need to be inserted between characters because the PDF format has no notion of the space character. A space is inserted if the characters are further apart than the *word_margin* (W in the figure). The *word_margin* is relative to the maximum width or height of the new character. Having a smaller *word_margin* creates smaller words. Note that the *word_margin* should at least be smaller than the *char_margin* otherwise none of the characters will be separated by a space.

The result of this stage is a list of lines. Each line consists of a list of characters. These characters are either original *LTChar* characters that originate from the PDF file, or inserted *LTAnno* characters that represent spaces between words or newlines at the end of each line.

Grouping lines into boxes

The second step is grouping lines in a meaningful way. Each line has a bounding box that is determined by the bounding boxes of the characters that it contains. Like grouping characters, pdfminer.six uses the bounding boxes to group the lines.

Lines that are both horizontally overlapping and vertically close are grouped. How vertically close the lines should be is determined by the *line_margin*. This margin is specified relative to the height of the bounding box. Lines are close if the gap between the tops (see L_1 in the figure) and bottoms (see L_2) in the figure) of the bounding boxes is closer together than the absolute line margin, i.e. the *line_margin* multiplied by the height of the bounding box.

The result of this stage is a list of text boxes. Each box consists of a list of lines.

Grouping textboxes hierarchically

The last step is to group the text boxes in a meaningful way. This step repeatedly merges the two text boxes that are closest to each other.

The closeness of bounding boxes is computed as the area that is between the two text boxes (the blue area in the figure). In other words, it is the area of the bounding box that surrounds both lines, minus the area of the bounding boxes of the individual lines.

Working with rotated characters

The algorithm described above assumes that all characters have the same orientation. However, any writing direction is possible in a PDF. To accommodate for this, pdfminer.six allows to detect vertical writing with the *detect_vertical* parameter. This will apply all the grouping steps as if the pdf was rotated 90 (or 270) degrees

References

1.4 API Reference

1.4.1 Command-line API

pdf2txt.py

dumppdf.py

1.4.2 High-level functions API

extract_text

`pdfminer.high_level.`**`extract_text`**(*pdf_file: Union[pathlib.PurePath, str, io.IOBase], password: str = '', page_numbers: Optional[Container[int]] = None, maxpages: int = 0, caching: bool = True, codec: str = 'utf-8', laparams: Optional[*pdfminer.layout.LAParams*] = None*) → str

Parse and return the text contained in a PDF file.

Parameters

- **`pdf_file`** – Either a file path or a file-like object for the PDF file to be worked on.
- **`password`** – For encrypted PDFs, the password to decrypt.
- **`page_numbers`** – List of zero-indexed page numbers to extract.
- **`maxpages`** – The maximum number of pages to parse
- **`caching`** – If resources should be cached
- **`codec`** – Text decoding codec
- **`laparams`** – An LAParams object from pdfminer.layout. If None, uses some default settings that often work well.

Returns a string containing all of the text extracted.

extract_text_to_fp

`pdfminer.high_level.`**`extract_text_to_fp`**`(`*inf: BinaryIO*, *outfp: Union[TextIO, BinaryIO]*, *output_type: str = 'text'*, *codec: str = 'utf-8'*, *laparams: Optional[pdfminer.layout.LAParams] = None*, *maxpages: int = 0*, *page_numbers: Optional[Container[int]] = None*, *password: str = ''*, *scale: float = 1.0*, *rotation: int = 0*, *layoutmode: str = 'normal'*, *output_dir: Optional[str] = None*, *strip_control: bool = False*, *debug: bool = False*, *disable_caching: bool = False*, ***kwargs: Any*`)` → None

Parses text from inf-file and writes to outfp file-like object.

Takes loads of optional arguments but the defaults are somewhat sane. Beware laparams: Including an empty LAParams is not the same as passing None!

Parameters

- **`inf`** – a file-like object to read PDF structure from, such as a file handler (using the builtin *open()* function) or a *BytesIO*.
- **`outfp`** – a file-like object to write the text to.
- **`output_type`** – May be ‘text’, ‘xml’, ‘html’, ‘tag’. Only ‘text’ works properly.
- **`codec`** – Text decoding codec
- **`laparams`** – An LAParams object from pdfminer.layout. Default is None but may not layout correctly.
- **`maxpages`** – How many pages to stop parsing after
- **`page_numbers`** – zero-indexed page numbers to operate on.
- **`password`** – For encrypted PDFs, the password to decrypt.
- **`scale`** – Scale factor
- **`rotation`** – Rotation factor
- **`layoutmode`** – Default is ‘normal’, see pdfminer.converter.HTMLConverter
- **`output_dir`** – If given, creates an ImageWriter for extracted images.
- **`strip_control`** – Does what it says on the tin
- **`debug`** – Output more logging data
- **`disable_caching`** – Does what it says on the tin
- **`other`** –

Returns nothing, acting as it does on two streams. Use StringIO to get strings.

extract_pages

pdfminer.high_level.**extract_pages**(*pdf_file: Union[pathlib.PurePath, str, io.IOBase], password: str = '', page_numbers: Optional[Container[int]] = None, maxpages: int = 0, caching: bool = True, laparams: Optional[*pdfminer.layout.LAParams*] = None*) → Iterator[pdfminer.layout.LTPage]

Extract and yield LTPage objects

Parameters

- **pdf_file** – Either a file path or a file-like object for the PDF file to be worked on.
- **password** – For encrypted PDFs, the password to decrypt.
- **page_numbers** – List of zero-indexed page numbers to extract.
- **maxpages** – The maximum number of pages to parse
- **caching** – If resources should be cached
- **laparams** – An LAParams object from pdfminer.layout. If None, uses some default settings that often work well.

Returns

1.4.3 Composable API

LAParams

class pdfminer.layout.**LAParams**(*line_overlap: float = 0.5, char_margin: float = 2.0, line_margin: float = 0.5, word_margin: float = 0.1, boxes_flow: Optional[float] = 0.5, detect_vertical: bool = False, all_texts: bool = False*)

Parameters for layout analysis

Parameters

- **line_overlap** – If two characters have more overlap than this they are considered to be on the same line. The overlap is specified relative to the minimum height of both characters.
- **char_margin** – If two characters are closer together than this margin they are considered part of the same line. The margin is specified relative to the width of the character.
- **word_margin** – If two characters on the same line are further apart than this margin then they are considered to be two separate words, and an intermediate space will be added for readability. The margin is specified relative to the width of the character.
- **line_margin** – If two lines are are close together they are considered to be part of the same paragraph. The margin is specified relative to the height of a line.
- **boxes_flow** – Specifies how much a horizontal and vertical position of a text matters when determining the order of text boxes. The value should be within the range of -1.0 (only horizontal position matters) to +1.0 (only vertical position matters). You can also pass *None* to disable advanced layout analysis, and instead return text based on the position of the bottom left corner of the text box.
- **detect_vertical** – If vertical text should be considered during layout analysis
- **all_texts** – If layout analysis should be performed on text in figures.

Todo:

- ***PDFDevice***
    - *TextConverter*
    - *PDFPageAggregator*
- *PDFPageInterpreter*

1.5 Frequently asked questions

1.5.1 Why is it called pdfminer.six?

Pdfminer.six is a fork of the original pdfminer created by Euske. Almost all of the code and architecture is in fact created by Euske. But, for a long time this original pdfminer did not support Python 3. Until 2020 the original pdfminer only supported Python 2. The original goal of pdfminer.six was to add support for Python 3. This was done with the six package. The six package helps to write code that is compatible with both Python 2 and Python 3. Hence, pdfminer.six.

As of 2020, pdfminer.six dropped the support for Python 2 because it was end-of-life. While the .six part is no longer applicable, we kept the name to prevent breaking changes for existing users.

The current punchline "We fathom PDF" is a whimsical reference to the six. Fathom means both deeply understanding something, and a fathom is also equal to six feet.

1.5.2 How does pdfminer.six compare to other forks of pdfminer?

Pdfminer.six is now an independent and community maintained package for extracting text from PDF's with Python. We actively fix bugs (also for PDF's that don't strictly follow the PDF Reference), add new features and improve the usability of pdfminer.six. This community separates pdfminer.six from the other forks of the original pdfminer. PDF as a format is very diverse and there are countless deviations from the official format. The only way to support all the PDF's out there is to have a community that actively uses and improves pdfminer.

Since 2020, the original pdfminer is dormant, and pdfminer.six is the fork which Euske recommends if you need an actively maintained version of pdfminer.

CHAPTER

TWO

FEATURES

- Parse all objects from a PDF document into Python objects.
- Analyze and group text in a human-readable way.
- Extract text, images (JPG, JBIG2 and Bitmaps), table-of-contents, tagged contents and more.
- Support for (almost all) features from the PDF-1.7 specification
- Support for Chinese, Japanese and Korean CJK) languages as well as vertical writing.
- Support for various font types (Type1, TrueType, Type3, and CID).
- Support for RC4 and AES encryption.
- Support for AcroForm interactive form extraction.

CHAPTER THREE

INSTALLATION INSTRUCTIONS

Before using it, you must install it using Python 3.6 or newer.

```
$ pip install pdfminer.six
```

Optionally install extra dependencies that are needed to extract jpg images.

```
$ pip install 'pdfminer.six[image]'
```

CHAPTER FOUR

CONTRIBUTING

We welcome any contributors to pdfminer.six! But, before doing anything, take a look at the contribution guide.

INDEX