pdfreader Documentation

Release 0.1.13dev

Maksym polshcha

Aug 11, 2022

Contents

CHAPTER 1

Overview

pdfreader is a Pythonic API to PDF documents which follows PDF-1.7 specification.

It allows to parse documents, extract texts, images, fonts, CMaps, and other data; access different objects within PDF documents.

Features:

- Extracts texts (plain and formatted)
- Extracts forms data (plain and formatted)
- Extracts images and image masks as Pillow/PIL Images
- Supports all PDF encodings, CMap, predefined cmaps.
- Browse any document objects, resources and extract any data you need (fonts, annotations, metadata, multimedia, etc.)
- Document history access and access to previous document versions if incremental updates are in place.
- Follows PDF-1.7 specification
- Fast document processing due to lazy objects access

Installing / Upgrading Instructions on how to get and install the distribution.

Tutorial A quick overview on how to start.

Examples and HowTos Examples of how to perform specific tasks.

pdfreader API API documentation, organized by module.

CHAPTER 2

Issues, Support and Feature Requests

If you're having trouble, have questions about *pdfreader*, or need some features the best place to ask is the Github issue tracker. Once you get an answer, it'd be great if you could work it back into this documentation and contribute!

CHAPTER 3

Contributing

pdfreader is an open source project. You're welcome to contribute:

- Code patches
- Bug reports
- Patch reviews
- Introduce new features
- Documentation improvements

pdfreader uses GitHub issues to keep track of bugs, feature requests, etc.

See project sources

CHAPTER 4

Donation

If this project is helpful, you can treat me to coffee :-)

CHAPTER 5

About This Documentation

This documentation is generated using the Sphinx documentation generator. The source files for the documentation are located in the *doc/* directory of the *pdfreader* distribution. To generate the docs locally run the following command from the root directory of the *pdfreader* source:

```
$ python setup.py doc
```

CHAPTER 6

Table of Contents

6.1 Installing / Upgrading

pdfreader is in the Python Package Index.

6.1.1 Installing with pip

We recommend using pip to install *pdfreader* on all platforms:

```
$ python -m pip install pdfreader
```

To get a specific version of pdfreader:

```
$ python -m pip install pdfreader==0.1.2
```

To upgrade using pip:

```
$ python -m pip install --upgrade pdfreader
```

6.1.2 Installing with easy_install

To install with `easy_install` from setuptools do:

```
$ python -m easy_install pdfreader
```

6.1.3 Installing from source

You can also download the project source and do:

```
$ git clone git://github.com/maxpmaxp/pdfreader.git pdfreader
$ cd pdfreader/
$ python setup.py install
```

6.1.4 Python versions support

pdfreader supports Python 3.6+. It might work on 3.4 and 3.5 but was never tested.

It is not compatible with Python 2.

6.2 Tutorial

Have a look at the `sample file`. In this tutorial we will learn simple methods on - how to open it - navigate pages - exract images and texts.

6.2.1 Prerequisites

Before we start, let's make sure that you have the *pdfreader* distribution *installed*. In the Python shell, the following should run without raising an exception:

```
>>> import pdfreader
>>> from pdfreader import PDFDocument, SimplePDFViewer
```

6.2.2 How to start

Note: If you need to extract texts/images or other content from PDF you can skip these chapters and go directly to *How to start extracting PDF content*.

The first step when working with *pdfreader* is to create a *`PDFDocument`* instance from a binary file. Doing so is easy:

```
>>> fd = open(file_name, "rb")
>>> doc = PDFDocument(fd)
```

As *pdfreader* implements lazy PDF reading (it never reads more then you ask from the file), so it's important to keep the file opened while you are working with the document. Make sure you don't close it until you're done.

It is also possible to use a binary file-like object to create an instance, for example:

```
>>> from io import BytesIO
>>> with open(file_name, "rb") as f:
...     stream = BytesIO(f.read())
>>> doc2 = PDFDocument(stream)
```

Let's check the PDF version of the document and it's metadata

```
>>> doc.header.version
'1.6'
>>> doc.metadata
{'CreationDate': datetime.datetime(2019, 10, 29, ... 'Producer': 'SAMBox 1.1.19 (www.
↪sejda.org)'}
```

Now we can go ahead to the document catalog and walking through pages.

6.2.3 How to access Document Catalog

Catalog (aka Document Root) contains all you need to know to start working with the document: metadata, reference to pages tree, layout, outlines etc.

```
>>> doc.root.Type
'Catalog'
>>> doc.root.Metadata.Subtype
'XML'
>>> doc.root.Outlines.First['Title']
b'Start of Document'
```

For the full list of document root attributes see PDF-1.7 specification section 7.7.2

6.2.4 How to browse document pages

There is a generator *pages()* to browse the pages one by one. It yields *Page* instances.

```
>>> page_one = next(doc.pages())
```

You may read all the pages at once

```
>>> all_pages = [p for p in doc.pages()]
>>> len(all_pages)
15
```

Now we know how many pages are there!

You may wish to get some specific page if your document contains hundreds and thousands. Doing this is just a little bit trickier. To get the 6th page you need to walk through the previous five.

```
>>> from itertools import islice
>>> page_six = next(islice(doc.pages(), 5, 6))
>>> page_five = next(islice(doc.pages(), 4, 5))
```

Don't forget, that all PDF viewers start page numbering from 1, however Python lists start their indexes from 0.

```
>>> page_eight = all_pages[7]
```

Now we can access all page attributes:

```
>>> page_six.MediaBox
[0, 0, 612, 792]
>>> page_six.Annots[0].Subj
b'Text Box'
```

It's possible to access parent Pages Tree Node for the page, which is *PageTreeNode* instance, and all it's kids:

```
>>> page_six.Parent.Type
'Pages'
>>> page_six.Parent.Count
15
>>> len(page_six.Parent.Kids)
15
```

Our example contains the only one Pages Tree Node. That is not always true.

For the complete list Page and Pages attributes see PDF-1.7 specification sections 7.7.3.2-7.7.3.3

6.2.5 How to start extracting PDF content

It's possible to extract raw data with *PDFDocument* instance but it just represents raw document structure. It can't interpret PDF content operators, that's why it might be hard.

Fortunately there is *SimplePDFViewer*, which understands a lot. It is a simple PDF interpreter which can "display" (whatever this means) a page on *SimpleCanvas*.

```
>>> fd = open(file_name, "rb")
>>> viewer = SimplePDFViewer(fd)
```

Document metadata is also accessible through *SimplePDFViewer* instance:

```
>>> viewer.metadata
{'CreationDate': datetime.datetime(2019, 10, 29, ... 'Producer': 'SAMBox 1.1.19 (www.
↪sejda.org)'}
```

The viewer instance gets content you see in your Adobe Acrobat Reader. *SimplePDFViewer* provides you with *SimpleCanvas* objects for every page. This object contains page content: images, forms, texts.

The code below walks through all document's pages and extracts data:

```
>>> for canvas in viewer:
...     page_images = canvas.images
...     page_forms = canvas.forms
...     page_text = canvas.text_content
...     page_inline_images = canvas.inline_images
...     page_strings = canvas.strings
>>>
```

Also you can navigate to some specific page with *navigate()* and call *render()*

```
>>> viewer.navigate(8)
>>> viewer.render()
>>> page_8_canvas = viewer.canvas
```

The viewer extracts:

- page images (XObject)
- page inline images (BI/ID/EI operators)
- page forms (XObject)
- decoded page strings (PDF encodings & CMap support)
- human (and robot) readable page markdown - original PDF commands containing decoded strings.

6.2.6 Extracting Page Images

There are 2 kinds of images in PDF documents:

- XObject images
- inline images

Every one is represented by its own class (`Image` and `InlineImage`)

Let's extract some pictures now! They are accessible through `canvas` attribute. Have a look at `page 8` of the sample document. It contains a fax message, and is is available on `inline_images` list.

```
>>> len(viewer.canvas.inline_images)
1
>>> fax_image = viewer.canvas.inline_images[0]
>>> fax_image.Filter
'CCITTFaxDecode'
>>> fax_image.Width, fax_image.Height
(1800, 3113)
```

This would be nothing if you can't see the image itself :-) Now let's convert it to a Pillow/PIL Image object and save!

```
>>> pil_image = fax_image.to_Pillow()
>>> pil_image.save('fax-from-p8.png')
```

Voila! Enjoy opening it in your favorite editor!

Check the complete list of Image (sec. 8.9.5) and InlineImage (sec. 8.9.7) attributes.

6.2.7 Extracting texts

Getting texts from a page is super easy. They are available on `strings` and `text_content` attributes.

Let's go to the previous page (`#7`) and extract some data.

```
>>> viewer.prev()
```

Remember, when you navigate another page the viewer resets the canvas.

```
>>> viewer.canvas.inline_images == []
True
```

Let's render the page and see the texts.

- Decoded plain text strings are on `strings` (by pieces and in order they come on the page)
- Decoded strings with PDF markdown are on `text_content`

```
>>> viewer.render()
>>> viewer.canvas.strings
['P', 'E', 'R', 'S', 'O', 'N', 'A', 'L', ... '2', '0', '1', '7']
```

As you see every character comes as an individual string in the page content stream here. Which is not usual.

Let's go to the very `first page`

```
>>> viewer.navigate(1)
>>> viewer.render()
>>> viewer.canvas.strings
[' ', 'P', 'l', 'a', 'i', 'nt', 'i', 'f', 'f', ... '10/28/2019 1:49 PM', '19CV47031']
```

PDF markdown is also available.

```
>>> viewer.canvas.text_content
"\n BT\n0 0 0 rg\n/GS0 gs... ET"
```

And the strings are decoded properly. Have a look at `the file`:

```
>>> with open("tutorial-sample-content-stream-p1.txt", "w") as f:
...     f.write(viewer.canvas.text_content)
19339
```

pdfreader takes care of decoding binary streams, character encodings, CMap, fonts etc. So finally you have human-readable content sources and markdown.

6.2.8 Hyperlinks and annotations

Let's Have a look at the `sample file`.

```
>>> fd = open(annotations_file_name, "rb")
>>> viewer = SimplePDFViewer(fd)
>>> viewer.navigate(1)
>>> viewer.render()
```

It contains several hyperlinks. Let's extract them!

Unlike HTML, PDF links are rectangle parts of viewing area, they are neither text properties nor attributes. That's why you can't find linked URLs in text content:

```
>>> plain_text = "".join(viewer.canvas.strings)
>>> "http" in plain_text
False
```

Links can be found in *:class:'~pdfreader.types.objects.Page* annotations (see 12.5 Annotations), which help user to interact with document.

Annotations for a current page are accessible through `annotations()`. The sample document has 3 annotations:

```
>>> len(viewer.annotations)
3
```

There are different types of annotations. Hyperlinks have *Subtype* of *Link*. We're ready to extract URLs:

```
>>> links = [annot.A.URI for annot in viewer.annotations
...          if annot.Subtype == 'Link']
>>> links
[b'http://www.apple.com', b'http://example.com', b'mailto:example@example.com']
```

6.2.9 Encrypted and password-protected PDF files

What if your file is protected by a password? Not a big deal! *pdfreader* supports encrypted and password-protected files! Just specify the password when create *`PDFDocument`* or *`SimplePDFViewer`*.

Let's see how this works with an encrypted password-protected file `sample file`. The password is *qwerty*.

```
>>> fd = open(protected_file_name, "rb")
>>> viewer = SimplePDFViewer(fd, password="qwerty")
>>> viewer.render()
>>> text = "".join(viewer.canvas.strings)
>>> text
'Sed ut perspiciatis unde omnis iste ... vel illum qui dolorem eum fugiat quo␣
↪voluptas nulla pariatur?'
```

The same about `PDFDocument`:

```
>>> fd = open(protected_file_name, "rb")
>>> doc = PDFDocument(fd, password="qwerty")
>>> page_one = next(doc.pages())
>>> page_one.Contents
<Stream:len=1488,data=b'...'>
```

What if the password is wrong? It throws an exception.

```
>>> fd = open(protected_file_name, "rb")
>>> doc = PDFDocument(fd, password="wrong password")
Traceback (most recent call last):
...
ValueError: Incorrect password
```

The same for `SimplePDFViewer`:

```
>>> fd = open(protected_file_name, "rb")
>>> doc = SimplePDFViewer(fd, password="wrong password")
Traceback (most recent call last):
...
ValueError: Incorrect password
```

Note: Do you know, that PDF format supports encrypted files protected by the default empty password? Despite the password is empty, such files are encrypted still. Fortunately, *pdfreader* detects end decrypts such files automatically, there is nothig special to do!

6.3 Examples and HowTos

Advanced PDF data extraction techniques with real-life examples.

PDFDocument vs. SimplePDFViewer

What is the difference? The usecases.

How to extract XObject or Inline Images, Image Masks

Instructions on how to extract different image types for further manipulations.

How to parse PDF texts

Advanced text objects access methods for further parsing.

How to parse PDF Forms

Instructions on how to extract text data from PDF Forms.

How to extract Font data from PDF

It’s possible to extract an embedded font. Let’s read how to do that.

How to extract CMap for a font from PDF

What if you need to see font’s CMap?

How to browse PDF objects

Instructions on how to navigate PDF documents and access it’s objects. Advanced techniques.

6.3.1 PDFDocument vs. SimplePDFViewer

***pdfreader* provides 2 different interfaces for PDFs:**

- *PDFDocument*
- *SimplePDFViewer*

What is the difference?

***PDFDocument*:**

- knows nothing about interpretation of content-level PDF operators
- knows all about PDF file and document structure (types, objects, indirect objects, references etc.)
- can be used to access any document object: XRef table, DocumentCatalog, page tree nodes (aka Pages), binary streams like Font, CMap, Form, Page etc.
- can be used to access raw objects content (raw page content stream for example)
- has no graphical state

***SimplePDFViewer*:**

- uses *PDFDocument* as document navigation engine
- can render document content properly decoding it and interpreting PDF operators
- has graphical state

Use *PDFDocument* to navigate document and access raw data.

Use *SimplePDFViewer* to extract content you see in your favorite viewer (Adobe Acrobat Reader, hehe :-).

Let's see several usecases.

How to extract XObject or Inline Images, Image Masks

Extracting Inline Images is discussed in tutorial *Extracting Page Images*, so let's focus on XObject Images and Image Masks.

Extracting XObject Image

Open a sample document.

```
>>> from pdfreader import PDFDocument
>>> fd = open(file_name, "rb")
>>> doc = PDFDocument(fd)
```

Have a look at the sample file `sample file`. There is a logo on the first page. Let's extract it.

```
>>> page = next(doc.pages())
```

Let's check a dictionary of XObject resources for the page:

```
>>> page.Resources.XObject
{'img0': <IndirectReference:n=11,g=0>}
```

This stands for an XObject named *img0*, and referenced under number 11 and generation 0. The object has not been read by *pdfreader* still. We are lazy readers. We read objects only when we need them. Let's see what the object is.

```
>>> xobj = page.Resources.XObject['img0']
```

We just read the object (*__getitem__* does this implicitly) and now we may access its attributes.

```
>>> xobj.Type, xobj.Subtype
('XObject', 'Image')
```

Wow! It's really an image. Should we care about it's internal PDF representation? Of course no, let's just convert it to a Pillow/PIL Image and save.

```
>>> pil_image = xobj.to_Pillow()
>>> pil_image.save("extract-logo.png")
```

And here we are!

POLICE OFFICER
POLICE
COMM. OF
MASSACHUSETTS
BILLERICA

Try to open it and see any differences. It's absolutely the same as in the document.

Now you can manipulate *pil_image* with usual PIL methods: rotate, convert, blur, split, inverse, merge and so on, so on, so on.

Extracting Images: a very simple way

A very simple way also exisits. Use `SimplePDFViewer`:

```
>>> from pdfreader import SimplePDFViewer
>>> fd = open(file_name, "rb")
>>> viewer = SimplePDFViewer(fd)
>>> viewer.render()
```

After rendering all 1st page images are on the canvas

```
>>> all_page_images = viewer.canvas.images
>>> all_page_inline_images = viewer.canvas.inline_images
>>> img = all_page_images['img0']
>>> img.Type, img.Subtype
('XObject', 'Image')
```

Now you can convert it with magic `to_Pillow()` method, save or do whatever you want!

Extracting Image Masks

Image Mask is just a specific kind of image actually. Except it is not always visible directly in your PDF Viewer. Nevertheless it can be accessed absolutely the same way.

Let's have a look at the `example` from *Extracting Page Images*, and see what image masks it contains.

```
>>> from pdfreader import SimplePDFViewer
>>> fd = open(pdf_file_name, "rb")
>>> viewer = SimplePDFViewer(fd)
```

We use *Image.ImageMask* attribute to filter image masks from another images. Let's go to the 5th page and take the first image mask:

```
>>> viewer.navigate(5)
>>> viewer.render()
>>> inline_images = viewer.canvas.inline_images
>>> image_mask = next(img for img in inline_images if img.ImageMask)
```

Now convert it to Pillow object and save:

```
>>> pil_img = image_mask.to_Pillow()
>>> pil_img.save("mask.png")
```

Have a look! What a beautiful QR-code!

Useful links

You find the complete list of PDF image attributes in the specification:

- Image (sec. 8.9.5)
- InlineImage (sec. 8.9.7)

How to parse PDF texts

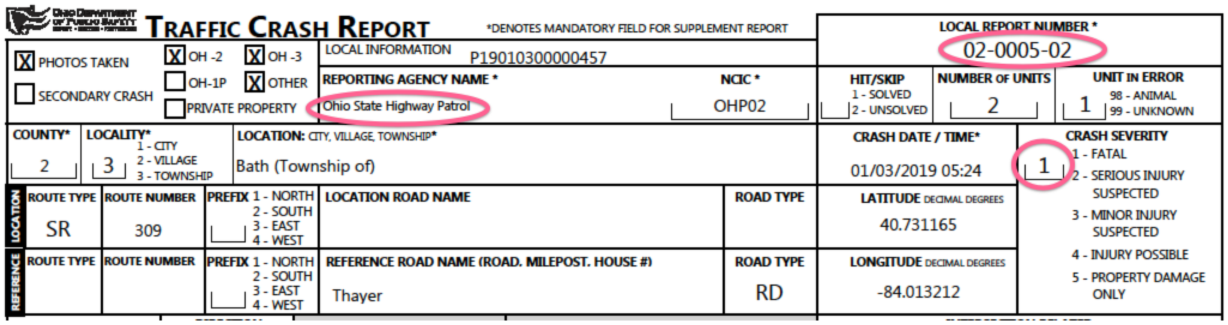
Simple ways of getting plain texts and formatted texts from documents are discussed in the tutorial *Extracting texts*, so let's focus on advanced techniques.

In this example we build a parser for `traffic crash reports`, that extracts:

- local report number
- reporting agency name

- crash severity

from the first page. The parser can be applied to all crash reports like that.



TRAFFIC CRASH REPORT *DENOTES MANDATORY FIELD FOR SUPPLEMENT REPORT

LOCAL REPORT NUMBER * 02-0005-02

[✓ PHOTOS TAKEN] [✓ OH -2] [✓ OH -3] LOCAL INFORMATION P19010300000457

SECONDARY CRASH OH-1P [✓ OTHER] REPORTING AGENCY NAME * Ohio State Highway Patrol NCIC * OHP02

PRIVATE PROPERTY

HIT/SKIP 1 - SOLVED 2 - UNSOLVED | NUMBER OF UNITS 2 | UNIT IN ERROR 1 98 - ANIMAL 99 - UNKNOWN

COUNTY* 2 | LOCALITY* 3 1 - CITY 2 - VILLAGE 3 - TOWNSHIP | LOCATION: CITY, VILLAGE, TOWNSHIP* Bath (Township of) | CRASH DATE / TIME* 01/03/2019 05:24 | CRASH SEVERITY 1 1 - FATAL 2 - SERIOUS INJURY SUSPECTED 3 - MINOR INJURY SUSPECTED 4 - INJURY POSSIBLE 5 - PROPERTY DAMAGE ONLY

LOCATION: ROUTE TYPE SR | ROUTE NUMBER 309 | PREFIX 1 - NORTH 2 - SOUTH 3 - EAST 4 - WEST | LOCATION ROAD NAME | ROAD TYPE | LATITUDE DECIMAL DEGREES 40.731165

REFERENCE: ROUTE TYPE | ROUTE NUMBER | PREFIX 1 - NORTH 2 - SOUTH 3 - EAST 4 - WEST | REFERENCE ROAD NAME (ROAD, MILEPOST, HOUSE #) Thayer | ROAD TYPE RD | LONGITUDE DECIMAL DEGREES -84.013212

Let's open the document and render the first page:

```
>>> from pdfreader import SimplePDFViewer
>>> fd = open(file_name, "rb")
>>> viewer = SimplePDFViewer(fd)
>>> viewer.render()
```

Every PDF page has one or more binary content streams associated with it. Streams may contain inline images, text blocks, text formatting instructions, display device operators etc. In this example we stay focused on text blocks.

Every text block in a stream is surrounded by BT/ET instructions and usually tricky encoded. Fortunately the viewer understands lot of PDF operators and encoding methods, so after rendering we may access human-readable PDF markup containing decoded strings.

```
>>> markdown = viewer.canvas.text_content
>>> markdown
"... BT\n/F3 6.0 Tf\n0 0 rg\n314.172 TL\n168.624 759.384 Td\n(LOCAL INFORMATION)␣
↪Tj\n ..."
```

This text block contains instructions for a viewer (font, positioning etc.) and one string surrounded by brackets.

```
>>> viewer.canvas.strings
['LOCAL INFORMATION', 'P19010300000457', ...]
```

Text-related `SimpleCanvas` attributes are:

- `text_content` - contains all data within a single BT/ET block: commands and text strings. All text strings are surrounded by brackets and decoded according to the current graphical state (*q, Q, gs, Tf* and few other commands). The value can be used to parse text content by PDF markdown.
- `strings` - list of all strings as they come in text blocks. Just decoded plain text. No PDF markdown here.

How to parse PDF markdown

At this point *markdown* contains all texts with PDF markdown from the page.

```
>>> isinstance(markdown, str)
True
```

Let's save it as a text file and analyze how can we extract the data we need.

```
>>> with open("example-crash-markdown.txt", "w") as f:
...     f.write(markdown)
52643
```

Open your favorite editor and have a look at `the file`.

Now we may use any text processing tools like regular expressions, grep, custom parsers to extract the data.

```
>>> reporting_agency = markdown.split('(REPORTING AGENCY NAME *)', 1)[1].split('(',␣
↪1)[1].split(')',1)[0]
>>> reporting_agency
'Ohio State Highway Patrol'

>>> local_report_number = markdown.split('(LOCAL REPORT NUMBER *)', 1)[1].split('(',␣
↪1)[1].split(')',1)[0]
>>> local_report_number
'02-0005-02'

>>> crash_severity = markdown.split('( ERROR)', 1)[1].split('(', 1)[1].split(')',1)[0]
>>> crash_severity
'1'
```

Here we are!

Useful links

- Detailed description of PDF texts is here (see sec. 9)
- Conforming reader graphical state reading is here (see sec. 8.4)

How to parse PDF Forms

In most cases texts come within page binary content streams and can be extracted as in *Extracting texts* and *How to parse PDF texts*.

There is one more place where text data can be found: page forms. Form is a special subtype of XObject which is a part of page resources, and can be referenced from page by *do* command.

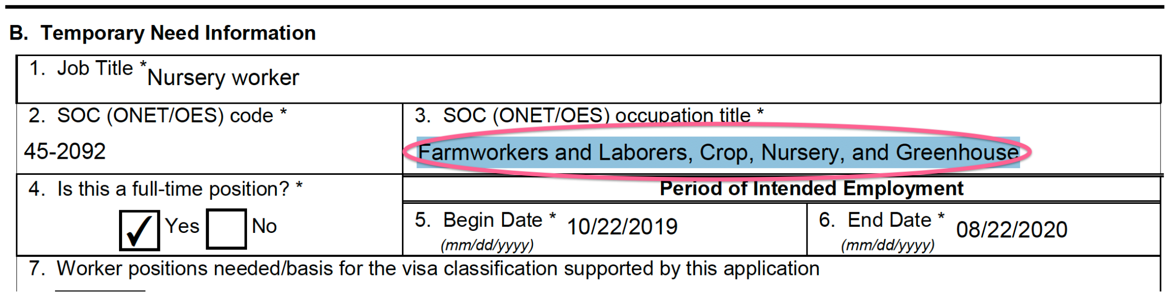
You may think of Form as of “small subpage” that is stored aside main content.

Have a look at one `PDF form`.

Let’s open the document and get the 1st page.

```
>>> from pdfreader import SimplePDFViewer
>>> fd = open(file_name, "rb")
>>> viewer = SimplePDFViewer(fd)
```

And now, let’s try to locate a string, located under the section *B.3 SOC (ONET/OES) occupation title*



B. Temporary Need Information

1. Job Title * Nursery worker	
2. SOC (ONET/OES) code * 45-2092	3. SOC (ONET/OES) occupation title * Farmworkers and Laborers, Crop, Nursery, and Greenhouse
4. Is this a full-time position? * [✓ Yes] No	Period of Intended Employment 5. Begin Date * (mm/dd/yyyy) 10/22/2019 — 6. End Date * (mm/dd/yyyy) 08/22/2020
7. Worker positions needed/basis for the visa classification supported by this application	

```
>>> viewer.render()
>>> plain_text = "".join(viewer.canvas.strings)
>>> "Farmworkers and Laborers" in plain_text
False
```

Apparently, the texts typed into the form are in some other place. They are in Form XObjects, listed under page resources. The viewer puts them on canvas:

```
>>> sorted(list(viewer.canvas.forms.keys()))
['Fm1', 'Fm10', 'Fm11', 'Fm12', 'Fm13', 'Fm14',...]
```

As Form is a kind of "sub-document" every entry in *viewer.canvas.forms* dictionary maps to `SimpleCanvas` instance:

```
>>> form9_canvas = viewer.canvas.forms['Fm9']
>>> "".join(form9_canvas.strings)
'Farmworkers and Laborers, Crop, Nursery, and Greenhouse'
```

Here we are!

More on PDF Form objects: see sec. 8.10

How to extract CMap for a font from PDF

In this example we extract CMap data for a font from PDF file.

CMaps (Character Maps) are text files used in PDF to map character codes to character glyphs in CID fonts. They come to PDF from PostScript.

Let's open a `sample document`.

```
>>> from pdfreader import PDFDocument
>>> fd = open(file_name, "rb")
>>> doc = PDFDocument(fd)
```

Now let's navigate to the 3rd page:

```
>>> from itertools import islice
>>> page = next(islice(doc.pages(), 2, 3))
```

and check page's fonts.

```
>>> page.Resources.Font
{'R11': <IndirectReference:n=153,g=0>, ... 'R9': <IndirectReference:n=152,g=0>}
>>> len(page.Resources.Font)
9
```

We see 9 different font resources. As *pdfreader* is a lazy reader the font data has not been read yet. We just see the names and the references to the objects.

Let's have a look at font named *R26*.

```
>>> font = page.Resources.Font['R26']
>>> font.Subtype, bool(font.ToUnicode)
('Type1', True)
```

It is PostScript Type1 font, and texts use CMap provided by *ToUnicode* attribute. Font's *ToUnicode* attribute contains a reference to the CMap file data stream:

```
>>> cmap = font.ToUnicode
```

Cmap file is a `StreamBasedObject` instance containing flate encoded binary stream.

```
>>> type(cmap)
<class 'pdfreader.types.objects.StreamBasedObject'>
>>> cmap.Filter
'FlateDecode'
```

that can be decoded by accessing `filtered`:

```
>>> data = cmap.filtered
>>> data
b'/CIDInit /ProcSet findresource ... end\n'
>>> with open("sample-cmap.txt", "wb") as f:
...     f.write(data)
229
```

Voila! *229* bytes written :-)

As it is a text file you can `open it` with your favorite text editor.

How to extract Font data from PDF

In this example we extract font data from a PDF file.

Let's open a `sample document`.

```
>>> from pdfreader import PDFDocument
>>> fd = open(file_name, "rb")
>>> doc = PDFDocument(fd)
```

Now let's see what fonts the very first page uses:

```
>>> page = next(doc.pages())
>>> sorted(page.Resources.Font.keys())
['T1_0', 'T1_1', 'T1_2', 'TT0', 'TT1']
```

We see 5 fonts named *T1_0*, *T1_1*, *T1_2*, *TT0* and *TT1*. As *pdfreader* is a lazy reader the font data has not been read yet. We just have the names and the references to the objects.

Let's have a look at font *T1_0*.

```
>>> font = page.Resources.Font['T1_0']
>>> font.Subtype, font.BaseFont, font.Encoding
('Type1', 'SCMYNU+TimesNewRomanPSMT', 'WinAnsiEncoding')
```

It is PostScript Type1 font, based on TimesNewRomanPSMT. Texts use *WinAnsiEncoding*, which is almost like python's *cp1252*.

Font's *FontDescriptor* contains a reference to the font file data stream:

```
>>> font_file = font.FontDescriptor.FontFile
```

The font file is a flate encoded binary stream *StreamBasedObject*

```
>>> type(font_file)
<class 'pdfreader.types.objects.StreamBasedObject'>
>>> font_file.Filter
['FlateDecode']
```

which can be decoded by accessing `filtered`

```
>>> data = font_file.filtered
>>> with open("sample-font.type1", "wb") as f:
...     f.write(data)
16831
```

Voila! *16831* bytes written :-)

How to browse PDF objects

There could be a reason when you need to access raw PDF objects as they are in the document. Or even get an object by its number and generation, which is also possible. Let's see several examples with *PDFDocument*.

Accessing document objects

Let's take a sample file from *How to access Document Catalog* tutorial. We already discussed there how to locate document catalog.

```
>>> from pdfreader import PDFDocument
>>> fd = open(file_name, "rb")
>>> doc = PDFDocument(fd)
>>> catalog = doc.root
```

To walk through the document you need to know object attributes and possible values. It can be found on PDF-1.7 specification. Then simply use attribute names in your python code.

```
>>> catalog.Type
'Catalog'
>>> catalog.Metadata.Type
'Metadata'
>>> catalog.Metadata.Subtype
'XML'
>>> pages_tree_root = catalog.Pages
>>> pages_tree_root.Type
'Pages'
```

Attribute names are cases sensitive. Missing or non-existing attributes have value of *None*

```
>>> catalog.type is None
True
```

(continues on next page)

(continued from previous page)

```
>>> catalog.Metadata.subType is None
True
>>> catalog.Metadata.UnkNown_AttriBute is None
True
```

If object is an array, access its items by index:

```
>>> first_page = pages_tree_root.Kids[0]
>>> first_page.Type
'Page'
>>> first_page.Contents.Length
3890
```

If object is a stream, you can get either raw data (deflated in this example):

```
>>> raw_data = first_page.Contents.stream
>>> first_page.Contents.Length == len(raw_data)
True
>>> first_page.Contents.Filter
'FlateDecode'
```

or decoded content:

```
>>> decoded_content = first_page.Contents.filtered
>>> len(decoded_content)
18428
>>> decoded_content.startswith(b'BT\n0 0 0 rg\n/GS0 gs')
True
```

All object reads are lazy. *pdfreader* reads an object when you access it for the first time.

Locate objects by number and generation

On the file structure level all objects have unique number an generation to identify them. To get an object by number and generation (for example to track object changes if incremental updates took place on file), just run:

```
>>> num, gen = 2, 0
>>> raw_obj = doc.locate_object(num, gen)
>>> obj = doc.build(raw_obj)
>>> obj.Type
'Catalog'
```

6.4 pdfreader API

6.4.1 pdfreader.document submodule

class pdfreader.document.**PDFDocument** (*fobj*, *password=''*)
Constructor method

root = None
references to document's Catalog instance

header = None
contains PDF file header data

trailer = None
contains PDF file trailer data

pages()
Yields document pages one by one.

Returns *Page* generator.

build(*obj*, *visited=None*, *lazy=True*)
Resolves all indirect references for the object.

Parameters

- **obj** (*one of supported PDF types*) – an object from the document
- **lazy** (*bool*) – don't resolve subsequent indirect references if True (default).
- **visited** – Shouldn't be used. Internal param containing already resolved objects to not fall into infinite loops

locate_object(*num*, *gen*)

6.4.2 pdfreader.viewer submodule

class pdfreader.viewer.**SimplePDFViewer**(**args*, ***kwargs*)
Simple PDF document interpreter (viewer).

- uses *PDFDocument* as document navigation engine
- renders document page content onto *SimpleCanvas*
- has graphical state

On initialization automatically navigates to the 1st page.

Parameters

- **fobj** – file-like object: binary file descriptor, BytesIO stream etc.
- **password** – Optional. Password to access PDF content.

current_page_number
Contains current page number

gss
Reflects current graphical state. *GraphicsStateStack* instance.

canvas
Current page canvas - *SimpleCanvas* instance

resources
Current page resources. *Resources* instance.

render()
Renders current page onto current canvas by interpreting content stream(s) commands. Charnges: graphical state, canvas.

navigate(*n*)
Navigates viewer to n-th page of the document. Side-effects: clears canvas, resets page resources, resets graphics state

Parameters **n** – page number. The very first page has number 1
Raises ***PageDoesNotExist*** – if there is no n-th page

next()
Navigates viewer to the next page of the document. Side-effects: clears canvas, resets page resources, resets graphics state

Raises `PageDoesNotExist` – if there is no next page

`prev``()`

Navigates viewer to the previous page of the document. Side-effects: clears canvas, resets page resources, resets graphics state

Raises `PageDoesNotExist` – if there is no previous page

`__iter__``()`

Returns document's canvas iterator.

`iter_pages``()`

Returns document's pages iterator.

class `pdfreader.viewer.`**`SimpleCanvas`**

Very simple canvas for PDF viewer: can contain page images (inline and XObject), strings, forms and text content.

`text_content`

Shall be a meaningful string representation of page content for further usage (decoded strings + markdown for example)

`strings`

Shall be al list of decoded strings, no PDF commands

`images`

Shall be dict of *name* -> `Image` *XObjects* rendered with *do* command

`inline_images`

Shall be list of `InlineImage` objects as they appear on page stream (BI/ID/EI operators)

`forms`

Shall be dict of *name* -> `SimpleCanvas` built from Form *XObjects* displayed with *do* command

class `pdfreader.viewer.`**`GraphicsState`**`(`***kwargs*`)`

Viewer's graphics state. See PDF 1.7 specification

sec. 8.4 - Graphics state

sec. 9.3 - Text State Parameters and Operators

Parameters **`kwargs`** – dict of attributes to set

`CTM`

current transformation matrix

`LW`

line width

`LC`

line cap

`LJ`

line join style

`ML`

miter limit

`D`

line dash

`RI`

color rendering intent

I
flatness tolerance

Font [font_name, font_size]
shall be a list if exists - [font_name, font_size] (Tf operator)

Tc
char spacing

Tw
word spacing

Tz
horizontlal scaling

TL
text leading

Tr
text rendering mode

Ts
text rise

class pdfreader.viewer.**GraphicsStateStack**
Graphics state stack. See PDF 1.7 specification sec. 8.4.2 - Graphics State Stack

save_state()
Copies current state and puts it on the top

restore_state()
Restore previously saved state from the top

class pdfreader.viewer.**Resources**(***kwargs*)
Page resources. See sec 7.8.3 Resource Dictionaries

class pdfreader.viewer.**PageDoesNotExist**
Exception. Supposed to be raised by PDF viewers on navigation to non-existing pages.

6.4.3 pdfreader.types submodule

class pdfreader.types.objects.**DictBasedObject**(*doc*, **args*, ***kwargs*)
Dictionary-based object. Automatically resolves indirect references on attributes/items access

class pdfreader.types.objects.**StreamBasedObject**(*doc*, *stream*)
Stream-based object. Automatically resolves indirect references on attributes access

class pdfreader.types.objects.**ArrayBasedObject**(*doc*, *lst*)
Array-based object. Automatically resolves indirect references on items access

class pdfreader.types.objects.**Catalog**(*doc*, **args*, ***kwargs*)
Dictionary based object. (Type = Catalog) See PDF 1.7 specification sec. 7.7.2 - DocumentCatalog

class pdfreader.types.objects.**PageTreeNode**(*doc*, **args*, ***kwargs*)
Dictionary based object. (Type = Pages) See PDF 1.7 specification sec. 7.7.3.2 - Page Tree Nodes

pages(*node=None*)
Yields tree node pages one by one.
Returns `Page` generator.

class pdfreader.types.objects.**Page**(*doc*, **args*, ***kwargs*)
Dictionary based Page object. (Type = Page) See PDF 1.7 specification sec. 7.7.3.3 - Page Objects

class pdfreader.types.objects.**Image**(*doc*, *stream*)
: Stream based XObject object. (Type = XObject, Subtype = Image) See PDF 1.7 specification sec. 8.9 - Images

  to_Pillow()
  : Converts image into PIL.Image object.
    Returns PIL.Image instance

class pdfreader.types.objects.**Form**(*doc*, *stream*)
: Stream based XObject object. (Type = XObject, Subtype = Form) See PDF 1.7 specification sec. 8.10 - Form XObjects

class pdfreader.types.objects.**XObject**(*doc*, *stream*)
: Stream based XObject object. (Type = XObject) See PDF 1.7 specification sec. 8.8 - External Objects

class pdfreader.types.content.**InlineImage**(*entries*, *data*)
: BI/ID/EI operators content.

  Inline image looks like a stream-based object but really it is not. We just follow Stream interface to have an option to interact with InlineImage the same way as with XObject/Image

  dictionary
  : key-value image properties

  data
  : bytes, encoded image stream

  to_Pillow()
  : Converts image into PIL.Image object.
    Returns PIL.Image instance

class pdfreader.types.content.**Operator**(*name*, *args*)
: Page content stream operator. For example: */F01 12 Tf*

  name
  : operator name

  args
  : list of operands

pdfreader.**version = '0.1.13dev'**
: package version

PDF data extraction, browsing objects:

pdfreader.**PDFDocument**
: Alias for *pdfreader.document.PDFDocument*

pdfreader.**SimplePDFViewer**
: Alias for *pdfreader.viewer.SimplePDFViewer*

Major classes used by *pdfreader.viewer.SimplePDFViewer*

- *pdfreader.viewer.SimpleCanvas*
- *pdfreader.viewer.Resources*
- *pdfreader.viewer.GraphicsState*
- *pdfreader.viewer.GraphicsStateStack*

Major classes and types:

- *pdfreader.types.objects.StreamBasedObject*

- *pdfreader.types.objects.DictBasedObject*
- *pdfreader.types.objects.ArrayBasedObject*
- *pdfreader.types.objects.Catalog*
- *pdfreader.types.objects.Page*
- *pdfreader.types.objects.PageTreeNode*
- *pdfreader.types.objects.Image*

Objects you face up in page/form content streams:

- *pdfreader.types.content.InlineImage*
- *pdfreader.types.content.Operator*

Python Module Index

Index