pdf4py

Release 0.1.0

Apr 19, 2020

Contents

In this page are collected a bunch of examples that will show you the correct use of *pdf4py*.

CHAPTER 1

PDF objects count

Suppose we want to know how many PDF (in use) objects are in a PDF file. Let's use the following snippet to find it out.

```
>>> import pdf4py.parser
>>> fp = open('tests/pdfs/0000.pdf', 'rb')
>>> parser = pdf4py.parser.Parser(fp)
>>> all_xref_entries = list(parser.xreftable)
>>> len(all_xref_entries)
119
>>> for x in all_xref_entries[:10]:
...     print(x)
...
XrefInUseEntry(offset=15, object_number=1, generation_number=0)
XrefInUseEntry(offset=525989, object_number=2, generation_number=0)
XrefInUseEntry(offset=63, object_number=3, generation_number=0)
XrefInUseEntry(offset=60167, object_number=4, generation_number=0)
XrefInUseEntry(offset=285, object_number=5, generation_number=0)
XrefInUseEntry(offset=38737, object_number=6, generation_number=0)
XrefInUseEntry(offset=36091, object_number=7, generation_number=0)
XrefInUseEntry(offset=21676, object_number=8, generation_number=0)
XrefInUseEntry(offset=4102, object_number=9, generation_number=0)
XrefInUseEntry(offset=5162, object_number=10, generation_number=0)
```

The special method *__iter__* called on *xreftable* returns a generator over the in-use and compressed objects references. To know how many of them there are one must iterates over the generator until it is exhausted. This is what *list* does: to collect all entries.

CHAPTER 2

Navigate the document structure

A PDF document is organized in a hierarchical structure made up of basic constructs such as dictionaries and references. This snippet shows how one can navigate such a structure to access content that defines a particular page.

```
>>> from pdf4py.parser import Parser
>>> fp = open('tests/pdfs/0000.pdf', 'rb')
>>> parser = Parser(fp)
>>> parser.trailer
{'Size': 120, 'Root': PDFReference(object_number=119, generation_number=0), 'Info':␣
↪PDFReference(object_number=114, generation_number=0), 'ID': [PDFHexString(value=b
↪'C49DFA7375A44BAA174802F645A8A459'), PDFHexString(value=b
↪'C49DFA7375A44BAA174802F645A8A459')]}
>>> root_ref = parser.trailer['Root']
>>> root_dict = parser.parse_reference(root_ref)
>>> root_dict
{'Type': 'Catalog', 'Pages': PDFReference(object_number=2, generation_number=0)}
>>> pages = parser.parse_reference(root_dict['Pages'])
>>> pages
{'Type': 'Pages', 'Count': 10, 'Kids': [PDFReference(object_number=23, generation_
↪number=0), PDFReference(object_number=31, generation_number=0), PDFReference(object_
↪number=49, generation_number=0), PDFReference(object_number=58, generation_
↪number=0), PDFReference(object_number=64, generation_number=0), PDFReference(object_
↪number=71, generation_number=0), PDFReference(object_number=87, generation_
↪number=0), PDFReference(object_number=94, generation_number=0), PDFReference(object_
↪number=104, generation_number=0), PDFReference(object_number=110, generation_
↪number=0)]}
>>> page_1 = parser.parse_reference(pages['Kids'][0])
>>> page_1
{'Type': 'Page', 'Parent': PDFReference(object_number=2, generation_number=0),
↪'Contents': PDFReference(object_number=24, generation_number=0), 'Resources':␣
↪PDFReference(object_number=27, generation_number=0), 'MediaBox': [0, 0, 595.276,␣
↪841.89]}
>>> contents = parser.parse_reference(page_1['Contents'])
>>> contents
PDFStream(dictionary={'Length': PDFReference(object_number=25, generation_number=0),
↪'Filter': 'FlateDecode'}, stream=<function Parser._stream_reader.<locals>.complete_
↪reader at 0x7f43b1c19d90>)
```

(continues on next page)

(continued from previous page)

```
>>> data = contents.stream()
>>> data
b'q\n/I0 Do\nQ\nq\n0.539 w\nBT\n/F0 11 Tf\n0 TL\n48 804.69 Td\n[(Proceedings 7th␣
↪Modelica Conference, Como, Italy)65(, Sep. 20-22, 2009)]TJ\nET\nQ\nq\n0.539 w\nBT\n/
↪F0 11 Tf\n0 TL\n48 35.8 Td\n[(\xa9 )18(The Modelica )55(Association,␣
↪2009)]TJ\nET\nQ\nq\n0.539 w\nBT\n/F0 11 Tf\n0 TL\n289.388 35.8␣
↪Td\n[(251)]TJ\nET\nQ\nq\n0.49 w\nBT\n/F0 10 Tf\n0 TL\n435.066 37 Td\n[(DOI: 10.3384/
↪ecp09430032)]TJ\nET\nQ\n'
```

In the last part can be seen a sequence of instructions that the rendering program has to execute to depict the page. That sequence can be tokenized using *pdf4py.parser.SequentialParser*.

```
>>> from pdf4py.parser import SequentialParser
>>> seq = SequentialParser(data)
>>> seq_iter = iter(seq)
>>> next(seq_iter)
PDFOperator(value='q')
>>> next(seq_iter)
'I0'
>>> next(seq_iter)
PDFOperator(value='Do')
>>> next(seq_iter)
PDFOperator(value='Q')
>>> next(seq_iter)
PDFOperator(value='q')
>>> next(seq_iter)
0.539
```

The user that wishes to interpret these commands to extract text, images, or other higher level information must use *pdf4py* to build a software module for that purpose.

CHAPTER 3

API documentation

Functions and classes a user of *pdf4py* can use in its projects are collected into a bunch of modules listed below. The most important one is *parser module* which defines the class *Parser*, a (almost) conforming PDF parser whose coverage of PDF syntax rules is illustrated into *PDF 1.7 standard coverage*.

3.1 List of modules

3.1.1 parser module

class pdf4py.parser.**Parser**(*source*, *password=None*)

Parse a PDF document to retrieve PDF objects composing it.

The constructor takes as argument an object *source*, the sequence of bytes the PDF document is encoded into. It can be of type *bytes*, *bytearray* or file pointer opened for reading in binary mode. Optionally, the second argument is the password to be provided if the document is protected through encryption (if encrypted with AESV3, the password is of type *str*, else *bytes*). For example,

```
>>> from pdf4py.parser import Parser
>>> with open('path/to/file.pdf', 'rb') as fp:
>>>     parser = Parser(fp)
```

Creates a new instance of *Parser*. The constructor reads the Cross Reference Table of the PDF document to retrieve the list of PDF objects that are present and parsable in the document. The Cross Reference Table is then available as attribute of the newly created *Parser* instance. For more information about the cross reference table, see the *XRefTable* documentation.

After the instantiation, *parser* will have a *XRefTable* instance associated to the attribute *xreftable*. To retrieve PDF objects pass entries in the table to the *Parser.parse_reference* method.

parse_reference

Parse and retrieve the PDF object *xref_entry* points to.

Notes

PDF objects are not parsed when an instance of *Parser* is being created. Instead, parsing occurs when this method is called. To avoid that the same object is being parsed too many times, a LRU cache is being used to keep in memory the last 256 parsed objects.

Parameters **reference** (*XrefInUseEntry or XrefCompressedEntry or PDFReference*) – An entry in the XRefTable or a PDFReference object pointing to a PDFObject within the file that has to be parsed.

Returns **obj** – The parsed PDF object.

Return type one of the types used to represent a PDF object.

Raises *ValueError* if *reference* object type is not a valid one.

class pdf4py.parser.**SequentialParser**(*source*, ***kwargs*)

Implements a parser that is able to parse a PDF objects by scanning the input bytes sequence.

In other words, objects are extracted in the order they appear in the stream. For this reason it is used to parse *Content Streams*.

Note that this class is not able to parse a complete PDF file since the process requires random access in the file to retrieve information when required (for example to resolve a reference pointing at the Integer holding the length of a stream). However, this class is used in defining the more powerful *Parser*.

The constructor that must be used by users takes a positional argument, *source*, being the source bytes stream. It can by a *byte*, *bytearray* or a file pointer opened in binary mode. Other keyword arguments are used internally in pdf4y, specifically by the *Parser* class.

parse_object(*obj_num: Optional[tuple] = None*)

Parse the next PDF object from the token stream.

Parameters **obj_num** (*tuple*) – Tuple *(seq, gen)*, *seq* and *gen* being the sequence and the generation number of the object that is going to be parsed respectively. These values are known when the parsing action is instructed after a XRefTable lookup. This parameter is used only by the *Parser* class when the PDF is encrypted.

Returns **obj** – The parsed PDF object.

Return type one of the PDF types defined in module *types*

class pdf4py.parser.**XRefTable**(*previous: pdf4py.parser.XRefTable*, *inuse_objects: dict*, *free_objects: set*, *compressed_objects: Optional[dict] = None*)

Implements the functionalities of a Cross Reference Table.

The Cross Reference Table (XRefTable) is the index of all the PDF objects in a PDF file. An object is uniquely identified with a tuple *(s, g)* where *s* is the sequence number and *g* is the generation number. There are mainly two types of entries in such table:

- *XrefInUseEntry* entries that represent objects that are part of the PDF document's current structure, and
- *tuple* entries pointing at *free objects*, objects that are no longer used (for example, they have been eliminated in a modification of the document).
- *XrefCompressedEntry* entries that are objects in use but stored in a compressed stream.

The listed three object types are to be used with the *Parser.parse_reference* class method to actually retrieve the associated object.

There are two main ways to query a *XRefTable* instance:

- Iterating over the instance itself to get references to *in use* and *compressed* objects (but *not* free objects).

- Accessing a particular entry using the square brackets. A bidimentional index is used, representing the sequence and generation numbers. This is because it implements the __getitem__ method that is used by the parser to look up objects if required during the parsing process.

previous
Points to the *XRefTable* instance that is associated to the */Prev* key in the trailer dictionary of the current cross-reference table.

3.1.2 types module

Defines custom Python classes used transversely within the library.

Amongst these definition are found Python representations for PDF Objects (section 7.3 of the Standard), Lexer's output tokens, and XRefTable entry types.

class pdf4py.types.**PDFDictDelimiter**(*value*)
[Internal] Represents tokens << and >>.

value
Alias for field number 0

class pdf4py.types.**PDFHexString**(*value*)
Represents the PDF Object 'Hexadecimal string'.

An hexadecimal string is used mainly to encode a small quantity of binary data. The sequence of hexadecimal digits are not decoded from ascii but stored directly as bytes in *value* attribute. This is so because you tipically want to pass that value to the *binascii.unhexlify* function.

value
Alias for field number 0

class pdf4py.types.**PDFIndirectObject**(*object_number*, *generation_number*, *value*)
Represents a PDF indirect object.

Attribute *value* contains the PDF object the indirect object structure wraps.

generation_number
Alias for field number 1

object_number
Alias for field number 0

value
Alias for field number 2

class pdf4py.types.**PDFKeyword**(*value*)
[Internal] Represents a keyword in the PDF grammar, for example `xref`.

value
Alias for field number 0

class pdf4py.types.**PDFLiteralString**(*value*)
Represents the PDF Object 'Literal string'.

A literal string is a sequence of ASCII characters. This is in theory, in practice there are so many PDF writers that store non ASCII strings using this object type that is best to leave the associated value in bytes and pass to the user the duty of choosing the right decoding scheme.

value
Alias for field number 0

class pdf4py.types.**PDFOperator**(*value*)
Represents an operator appearing in a ContentStream.

value
Alias for field number 0

class pdf4py.types.**PDFReference**(*object_number*, *generation_number*)
Represent a PDF reference to a PDF Indirect object.

generation_number
Alias for field number 1

object_number
Alias for field number 0

class pdf4py.types.**PDFSingleton**(*value*)
[Internal] Represents a singleton in the PDF greammar, for example `{`.

value
Alias for field number 0

class pdf4py.types.**PDFStream**(*dictionary*, *stream*)
Represents a PDF stream.

The attribute *dictionary* points to the stream dictionary. The attribute *stream* is a callable object requiring no arguments that when called returns the stream content bytes. The content is read from the source only when *stream* is called, following the lazy loading philosophy around which pdf4py is built around.

dictionary
Alias for field number 0

stream
Alias for field number 1

class pdf4py.types.**PDFStreamReader**(*value*)
[Internal] A wrapper around a function `f(length)` returned by *Lexer* to *Parser* when parsing a PDF stream object.‘‘

value
Alias for field number 0

class pdf4py.types.**XrefCompressedEntry**(*object_number*, *objstm_number*, *index*)
Represents an entry in the Cross Reference Table pointing to an object that currently contributes to the final PDF render, but stored in a compressed object stream to reduce the size of the PDF file.

index
Alias for field number 2

object_number
Alias for field number 0

objstm_number
Alias for field number 1

class pdf4py.types.**XrefInUseEntry**(*offset*, *object_number*, *generation_number*)
Represents an entry in the Cross Reference Table pointing to an object that currently contributes to the final PDF render (as opposite to removed, i.e. *free*, objects).

generation_number
Alias for field number 2

object_number
Alias for field number 1

offset
Alias for field number 0

3.1.3 exceptions module

exception pdf4py.exceptions.**PDFGenericError**
Raised when a generic error happens.

exception pdf4py.exceptions.**PDFLexicalError**
Raised when a lexical error is encountered during input scanning.

exception pdf4py.exceptions.**PDFSyntaxError**
Raised when the parsed PDF does not conform to syntax rules.

exception pdf4py.exceptions.**PDFUnsupportedError**
Raised when the parser does not support a PDF feature.

exception pdf4py.exceptions.**PDFWrongPasswordError**
Raised when the user gives in input a wrong password.

CHAPTER 4

PDF 1.7 standard coverage

In this file the progress in implementing all the features in the PDF 1.7 standard is tracked. Chapters 1 to 6 of the standard are devoted to give a general introduction to the standard whereas Chapter 7 is where the PDF syntax is defined. It follows that the best way to keep track of the progress is to specify for each section whether the illustrated features have been implemented or not. As the development goes on, the various sections describing features that have been supported will be marked with an check symbol (✓) in the following table. Moreover, the tilde symbol (~) means almost every aspect is supported or that the implementation seems to work but more testing is necessary. Finally, the cross symbol () informs that there is no support at this stage for the associated feature.

Section	Description	Status
7.2	*Lexical conventions*	✓
7.3	*Objects*	~
7.3.2	Boolean objects	✓
7.3.3	Numeric objects	✓
7.3.4	String objects	✓
7.3.5	Name objects	✓
7.3.6	Array objects	✓
7.3.7	Dictionary objects	✓
7.3.8	Stream objects	~ (F parameter not supported yet)
7.3.9	Null object	✓
7.3.10	Indirect objects	✓
7.4	*Filters*	~
7.4.2	ASCIIHexDecode	~ (Testing is missing still)
7.4.3	ASCII85Decode	✓
7.4.4	LZWDecode	
7.4.4	FlateDecode	~ (Predictors must still be tested)
7.4.5	RunLengthDecode	✓
7.4.6	CCITTFaxDecode	~ (data returned ‘as is’)
7.4.7	JBIG2Decode	~ (data returned ‘as is’)
7.4.8	DCTDecode	~ (data returned ‘as is’)
7.4.9	JPXDecode	~ (data returned ‘as is’)

Continued on next page

Table 1 – continued from previous page

Section	Description	Status
7.4.10	Crypt	✓
7.5	*File Structure*	~
7.5.2	File header	✓
7.5.4	Cross Reference Table	✓
7.5.5	File trailer	✓
7.5.6	Incremental updates	✓
7.5.7	Object streams	✓
7.5.8	Cross Reference Streams	✓
7.6	*Encryption*	~ (no File Specs and Public Key Crypto)
7.6.1	General	✓
7.6.2	General Encryption Algorithm	✓
7.6.3	Standard Security Handler	~ (permission bits ignored)
7.6.4	Public Key Security Handler	
7.6.5	Crypt Filters	✓
7.7	*Document Structure*	
7.7.2	Document Catalog	
7.7.3	Page Tree	
7.8	*Content Streams and Resources*	
7.8.2	Content Streams	
7.8.3	Resource Dictionaries	
7.9	*Common Data Structures*	
7.9.2	String Object Types	
7.9.3	Text Streams	
7.9.4	Dates	
7.9.5	Rectangles	
7.9.6	Name Trees	
7.9.7	Number Trees	
7.10	*Functions*	
7.10.1	General	
7.10.1	Resource Dictionaries	
7.10.2	Type 0 (Sampled)	
7.10.3	Type 2 (Exponential Interp.)	
7.11	*File Specification*	
7.11.2	File Specification Strings	
7.11.3	File Specification Dictionaries	
7.11.4	Embedded File Streams	
7.11.5	URL Specifications	
7.11.6	Collection Items	
7.11.7	Maintenance of File Spec.	
7.12	*Extensions Dictionary*	
7.12.2	Developer Extensions Dictionary	
7.12.3	BaseVersion	
7.12.4	ExtensionLevel	

Subsequent chapters describe higher level aspects that are built on top of the PDF syntax and elementary objects. As of now there is no support for those features, as explained in the landing page of the documentation.

In addition, the AESV3 encryption method specified in the PDF 1.7 Extension 3 document has been implemented.

CHAPTER 5

pdf4py's documentation

The package *pdf4py* allows the user to analyze a PDF file at a very low level and in a very flexible way by giving access to its atomic components, the PDF objects. All through a very simple API that can be used to build higher level functionalities (e.g. text and/or image extraction). In particular, it defines the class *Parser* that reads the *Cross Reference Table* of a PDF document and uses its entries to give the user the ability to locate PDF objects within the file and parse them into suitable Python objects.

DISCLAIMER: this package hasn't reached a stable version (>= 1.0.0) yet. Although the parser API is quite simple it may change suddenly from one release to the next one. All breaking changes will be properly notified in the release notes.

5.1 Quick example

Here is a quick demonstration on how to use pdf4py. For more examples, look at the *Tutorials and examples* page.

```
>>> from pdf4py.parser import Parser
>>> fp = open('tests/pdfs/0000.pdf', 'rb')
>>> parser = Parser(fp)
>>> info_ref = parser.trailer['Info']
>>> print(info_ref)
PDFReference(object_number=114, generation_number=0)
>>> info = parser.parse_reference(info_ref)
>>> print(info)
{'Creator': PDFLiteralString(value=b'PaperCept Conference Management System'),
    ... , 'Producer': PDFLiteralString(value=b'PDFlib+PDI 7.0.3 (Perl 5.8.0/Linux)')}
>>> creator = info['Creator'].value.decode('utf8')
```

(continues on next page)

(continued from previous page)

```
>>> print(creator)
PaperCept Conference Management System
```

5.2 Extracting text or images

Extracting text from a PDF and other higher level analysis tasks are not natively supported as of now because of two reasons:

- their complexity is not trivial and would require a not indifferent amount of work which now I prefer investing into developing a complete and reliable parser;
- they are conceptually different tasks from PDF parsing, since the PDF does not define the concept of document as a sequence of paragraphs, images, and other objects that can be normally considered *content*.

Therefore, they require a separate implementation built on top of *pdf4py*. In don't exclude that in future these functionalities will be made available as modules in this package, but I am not planning to do it anytime soon.

5.3 Installation

You can install *pdf4py* using pip:

```
python3 -m pip install pdf4py
```

or download the latest release from GitHub and use the *setup.py* script.

5.4 Release Notes

You can find the list of all releases with associated notes on GitHub.

5.5 Why this package

One day at work I was asked to analyze some PDF files. To my surprise I had discovered that there was not an established Python module to easily parse a PDF document. In order to understand why I delved into the PDF 1.7 specification: since that moment I've got interested more and more in the inner workings of one of the most important and ubiquitous file format. And what's a better way to understand the PDF than writing a parser for it?

Python Module Index

p

Index