PDF Parser

Nov 07, 2022

Contents:

CHAPTER 1

Overview

1.1 Introduction

This PDF Parser is a tool built on top of PDF Miner to help extracting information from PDFs in Python. The main idea was to create a tool that could be driven by code to interact with the elements on the PDF and slowly classify them by creating sections and adding tags to them. It also comes with a helpful visualisation tool which enables you to examine the current status of your elements.

This page gives a brief overview of the PDF Parser, but there is also a full *Reference* of all the functionality. You may get a more in-depth overview by looking at the *Examples*.

1.2 Setup

You will need to have Python 3.6 or greater installed, and if you're installing the development requirements to use the visualise tool you will also need tkinter installed on your system. For information on how to do this, see https://tkdocs.com/tutorial/install.html.

We recommend you install the development requirements with `pip3 install py-pdf-parser[dev]`, which enables the visualise tool. If you don't need the visualise tool (for example in a production app once you've written your parsing scripts) you can simply run `pip3 install py-pdf-parser`.

1.3 When Should I Use Py PDF Parser?

Py PDF Parser is best suited to locating and extracting specific data in a structured way from a PDF. You can locate contents however you want (by text, location, font, etc), and since it is code-driven you have the flexibility to implement custom logic without having to deal with the PDF itself. Py pdf parser helps to abstract away things like page breaks (unless you want to use them), which helps to write robust code which will extract data from multiple PDFs of the same type, even if there are differences between each individual document.

Py PDF Parser is good at extracting tables in PDFs, and allows you to write code to programmatically locate the tables to extract. Page breaks (and even headers or footers) half way through your table can be ignored easily. If you're trying to extract all tables from a PDF, other tools (e.g. https://camelot-py.readthedocs.io/en/master/) are available and may be more appropriate.

If you're simply trying to extract all of the text from a PDF, other tools (e.g. https://textract.readthedocs.io/en/stable/python_package.html) may be more appropriate. Whilst you can still do this with Py PDF Parser, it is not designed to be a tool where you simply plug in a PDF and it spits it out in text format. Py PDF Parser is not a plug-and-play solution, but rather a tool to help you write code that extracts certain pieces of data from a structured PDF.

1.4 Loading A PDF

To load a PDF, use the `load_file()`: function from the *Loaders*. You will need to use `load_file()`: with a file path to be able to use the visualisation tool with your PDF as the background. If you don't have this, you can instead use the `load()`: function, but when you use the visualisation tool there will be no background.

We order the elements in a pdf, left-to-right, top-to-bottom. At the moment, this is not configurable. Each `PDFElement` within the `PDFDocument` are aware of their position, both on the page and within the document, and also have properties allowing you to access their font and text. For more information about `PDFDocument` and `PDFElement`, see *Components*.

Pay particular attention to the `la_params` argument. These will need to be fine-tuned for your PDF. We suggest immediately visualising your PDF using the visualisation tool to see how the elements have been grouped. If multiple elements have been counted as one, or vice versa, you should be able to fix this by tweaking the `la_params`.

1.5 Filtering

Once you have loaded your PDF, say into a variable `document`, you can start interacting with the elements. You can access all the elements by calling `document.elements`. You may now want to filter your elements, for example you could do `document.elements.filter_by_text_equal("foo")` to filter for all elements which say "foo". To view all available filters, have a look at the *Filtering* reference.

The `document.elements` object, and any filtered subset thereof, will be an `ElementList`. These act like sets of elements, and so you can union (`|`), intersect (`&`), difference (`-`) and symmetric difference (`^`) different filtered sets of elements.

You can also chain filters, which will do the same as intersecting multiple filters, for example `document.elements.filter_by_text_equal("foo").filter_by_tag("bar")` is the same as `document.elements.filter_by_text_equal("foo") & document.elements.filter_by_tag("bar")`.

If you believe you have filtered down to a single element, and would like to examine that element, you can call `extract_single_element()`. This will return said element, or raise an exception if there is not a single element in your list.

You can see an example of filtering in the *Simple Memo* example.

1.6 Classifying Elements

There are three ways to classify elements:

- add tags
- create sections

- mark certain elements as ignored

To add a tag, you can simply call *add_tag()* on an *PDFElement*, or *add_tag_to_elements()* on an *ElementList*. You can filter by tags.

To create a section, you can call *create_section()*. See *Sectioning* for more information and the *Order Summary* example for an example. When you create a section you simply specify a name for the section, and the start and end element for the section. Any elements between the start and end element will be included in your section. You can add multiple sections with the same name, and internally they will be given unique names. You can filter by either the non-unique `section_name`, or by the unique sections. Elements can be in multiple sections.

To mark an element as ignored, simply set the `ignore` property to `True`. Ignored elements will not be included in any *ElementList*, however existing lists which you have assigned to variables will not be re-calculated and so may still include the ignored elements.

To process a whole pdf, we suggest that you mark any elements you're not interested in as ignored, group any elements which are together into sections, and then add tags to important elements. You can then loop through filtered sets of elements to extract the information you would like.

1.7 Visualisation Tool

The PDF Parser comes with a visualisation tool. See the *Visualise* documentation. When you visualise your *PDFDocument*, you'll be able to see each page of the document in turn, with every *PDFElement* highlighted. You can hover over the elements to see their sections, tags and whether they are ignored or not. This is very helpful for debugging any problems.

You can use the arrow key icons to change page, and can press home to return to page 1. You can also use the scroll wheel on your mouse to zoom in and out.

You can see an example of the visualisation in the *Simple Memo* and *Order Summary* examples.

1.8 Font Mappings

You can filter elements by font. The font will be taken from the PDF itself, however often they have long and confusing names. You can specify a `font_mapping` when you load the document to map these to more memorable names. This `font_mapping` can either be a regex pattern or an exact string mapping. See the *Components* reference for the *PDFDocument* arguments for more information.

You can see an example of font mapping in the *Order Summary* example.

1.9 Tables

We have many functions to help extract tables. All of these use the positioning of the elements on the page to do this. See the *Tables* reference, and the *Order Summary* and *More Tables* examples.

CHAPTER 2

Examples

Below you can find links to the following examples:

- The *Simple Memo* example shows the very basics of using py-pdf-parser. You will see how to load a pdf document, start filtering the elements, and extract text from certain elements in the document.
- The *Order Summary* example explains how to use font mappings, sections, and how to extract simple tables.
- The *More Tables* example explains tables in more detail, showing how to extract more complex tables.
- The *Element Ordering* example shows how to specify different orderings for the elements on a page.
- The *Extracting Text From Figures* example shows how to extract text from figures.

2.1 Simple Memo

Our first example will be extracting information from a simple memo.

You can `download the example memo here`.

We will assume that your company issues these memos always in a consistent format, i.e. with the "TO", "FROM", "DATE", and "SUBJECT" fields, the main content of the memo. We would like to write some code such that we can extract the information from each memo.

2.1.1 Step 1 - Load the file

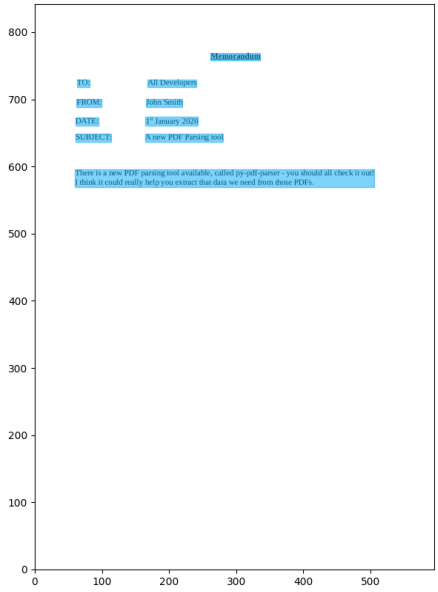
First, we should load the file into a *`PDFDocument`*, using *`load_file()`*:

```
from py_pdf_parser.loaders import load_file

document = load_file("simple_memo.pdf")
```

To check the PDF loaded as expected, we can use the *`visualise()`* tool by running

```
from py_pdf_parser.visualise import visualise

visualise(document)
```

This will open a matplotlib window which should look something like the following image:



Py-pdf-parser has extracted each element from the PDF as a *PDFElement*, and is showing a blue box around each element. This is what we are looking for. Always check the visualise tool, since sometimes you will need to adjust the layout parameters so that the tool correctly identifies your elements. We will get on to this in later examples.

2.1.2 Step 2 - Extract reference elements

Certain elements should be present in every memo. We will use these as reference elements to identify the elements which contain the information we are interested in. We already have our `document`, which is a *PDFDocument*. We can do *document.elements* to get a list (an *ElementList*) of all the *PDFElement* in the document, and also to allow us to filter the elements.

The simplest way to extract the elements we are interested in is by text. There are many other options available to us, and a full list can be found on the *filtering reference page*.

We will extract the "TO:", "FROM:", "DATE:" and "SUBJECT:" elements as reference elements, i.e. the elements on the left of the below image. We will then search to the right of each of them in turn, to extract the values for each field.

TO: All Developers

FROM: John Smith

DATE: 1st January 2020

SUBJECT: A new PDF Parsing tool

To extract the element which says "TO:", we can simply run *document.elements.filter_by_text_equal("TO:")*. This returns a new *ElementList* which contains all the elements in the document with text equal to "TO:". In this case, there should only be one element in the list. We could just use `[0]` on the element list to access the element in question, however, there is a convenience function, *extract_single_element()* on the *ElementList* class to handle this case. This essentially checks if the list has a single element and returns the element for you, otherwise it raises an exception. Use of this is encouraged to make your code more robust and to make any errors more explicit.

```
to_element = document.elements.filter_by_text_equal("TO:").extract_single_element()
from_element = document.elements.filter_by_text_equal("FROM:").extract_single_
↪element()
date_element = document.elements.filter_by_text_equal("DATE:").extract_single_
↪element()
subject_element = document.elements.filter_by_text_equal(
    "SUBJECT:"
).extract_single_element()
```

Each of the above elements will be a *PDFElement*.

2.1.3 Step 3 - Extract the data

In the above section we have extracted our reference elements. We can now use these to do some more filtering to extract the data we want. In particular, we can use *to_the_right_of()*, which will extract elements directly to the right of a given element. It effectively draws a dotted line from the top and bottom of your element out to the right hand side of the page, and any elements which are partially within the box created by the dotted line will be returned. To extract the text from a *PDFElement*, we must also call *.text()*.

```
to_text = document.elements.to_the_right_of(to_element).extract_single_element().
↪text()
from_text = (
    document.elements.to_the_right_of(from_element).extract_single_element().text()
)
date_text = (
    document.elements.to_the_right_of(date_element).extract_single_element().text()
)
subject_text_element = document.elements.to_the_right_of(
    subject_element
).extract_single_element()
subject_text = subject_text_element.text()
```

Note we keep a reference to the subject text element. This is because we will use it later.

We have now extracted the data from the top of the memo, for example `to_text` will be `"All Developers"`. The code does not rely on who the memo is to, and so it should still work for a memo with different values.

The last thing we need to do is extract the content of the memo. In our example there is only one paragraph, and so only one element, but if there were multiple paragraphs there could be multiple elements. There are a few ways to do this. It is probably the case that all the content elements are below the "SUBJECT:" element, however if the text started too far to the right this may not be the case. Instead, we can just use *`after()`* to filter for elements strictly after the `subject_text_element`:

```
content_elements = document.elements.after(subject_element)
content_text = "\n".join(element.text() for element in content_elements)
```

That is now everything extracted from the memo. We can wrap our output into any data structure we fancy, for example json:

```
output = {
    "to": to_text,
    "from": from_text,
    "date": date_text,
    "subject": subject_text,
    "content": content_text,
}
```

2.1.4 Full Code

Here is the full script constructed above:

```
from py_pdf_parser.loaders import load_file

# Step 1 - Load the document
document = load_file("simple_memo.pdf")

# We could visualise it here to check it looks correct:
# from py_pdf_parser.visualise import visualise
# visualise(document)

# Step 2 - Extract reference elements:
to_element = document.elements.filter_by_text_equal("TO:").extract_single_element()
from_element = document.elements.filter_by_text_equal("FROM:").extract_single_
↪element()
date_element = document.elements.filter_by_text_equal("DATE:").extract_single_
↪element()
subject_element = document.elements.filter_by_text_equal(
    "SUBJECT:"
).extract_single_element()

# Step 3 - Extract the data
to_text = document.elements.to_the_right_of(to_element).extract_single_element().
↪text()
from_text = (
    document.elements.to_the_right_of(from_element).extract_single_element().text()
)
date_text = (
    document.elements.to_the_right_of(date_element).extract_single_element().text()
)
subject_text_element = document.elements.to_the_right_of(
```

(continues on next page)

(continued from previous page)

```
    subject_element
).extract_single_element()
subject_text = subject_text_element.text()

content_elements = document.elements.after(subject_element)
content_text = "\n".join(element.text() for element in content_elements)

output = {
    "to": to_text,
    "from": from_text,
    "date": date_text,
    "subject": subject_text,
    "content": content_text,
}
```

This gives:

```
>>> from pprint import pprint
>>> pprint(output)

{'content': 'A new PDF Parsing tool\n'
            'There is a new PDF parsing tool available, called py-pdf-parser - '
            'you should all check it out!\n'
            'I think it could really help you extract that data we need from '
            'those PDFs.',
 'date': '1st January 2020',
 'from': 'John Smith',
 'subject': 'A new PDF Parsing tool',
 'to': 'All Developers'}
```

2.2 Order Summary

In this example we will extract some tabular data from an order summary pdf.

You can `download the example here`.

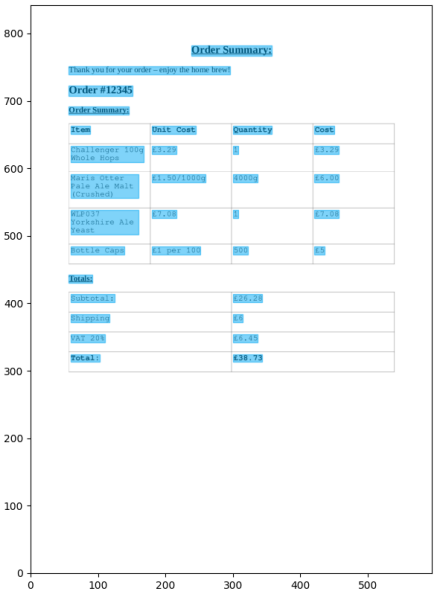
This is a fairly simple PDF, and as such it would be fairly easy to identify the tables and extract the data from them, however we will use this example to introduce font mappings and sections, which will come in useful for larger PDFs.

2.2.1 Step 1 - Load the file

We can *`load`* the file as follows, and take a quick look using the *`visualise tool`* to check it looks good.

```
from py_pdf_parser.loaders import load_file
from py_pdf_parser.visualise import visualise

document = load_file("order_summary.pdf")
visualise(document)
```

This should show the following. We should check that py-pdf-parser has detected each element correctly, which in this case it has.

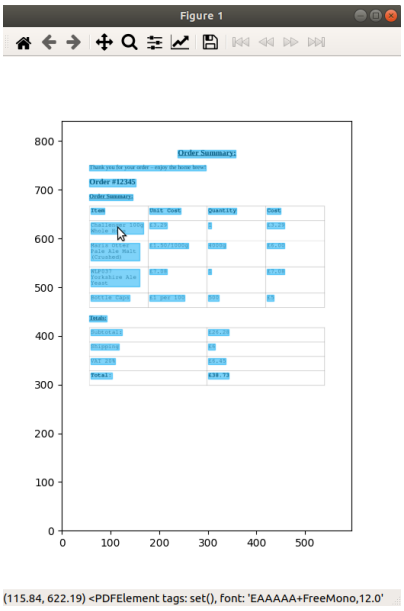


2.2.2 Step 2 - Use a font mapping

Each *PDFElement* has a *font* property, which is the name of the font in the PDF document (including the font size). You can use fonts to help filter elements.

Fonts often have long, not very useful names. However, additional keyword arguments passed to *load_file()* will be used to initialise the *PDFDocument*. One of these is the font mapping, which allows you to map the fonts in your PDF to more useful names.

The visualise tool allows you to inspect fonts. If you however over an element, a summary will be shown in text at the bottom of the window. For example, in the image below we hover over the first cell in the table, and can see that the font is `EAAAA+FreeMono,12.0`.



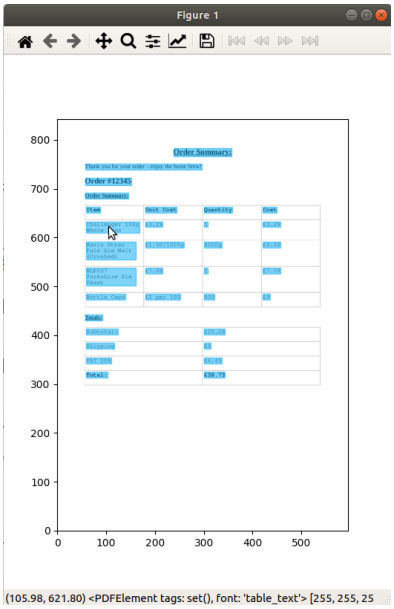
We can easily ask to see all of the available fonts by running

```
>>> set(element.font for element in document.elements)
{'EAAAAA+FreeMono,12.0', 'BAAAAA+LiberationSerif-Bold,16.0', 'CAAAAA+LiberationSerif,
↪12.0', 'DAAAAA+FreeMonoBold,12.0', 'BAAAAA+LiberationSerif-Bold,12.0'}
```

Using this and the visualise tool, we can now choose better names for each of the fonts, and then load the document again, but this time providing a font mapping.

```
FONT_MAPPING = {
    "BAAAAA+LiberationSerif-Bold,16.0": "title",
    "BAAAAA+LiberationSerif-Bold,12.0": "sub_title",
    "CAAAAA+LiberationSerif,12.0": "text",
    "DAAAAA+FreeMonoBold,12.0": "table_header",
    "EAAAAA+FreeMono,12.0": "table_text",
}
document = load_file("order_summary.pdf", font_mapping=FONT_MAPPING)
```

Using the visualise tool again, we can now see that our element's font has changed to `table_text`, which is a much more useful name for us.



2.2.3 Step 3 - Use regex for font mapping

In certain use cases (especially when handling many PDF files) you may encounter the problem that the same fonts have different prefixes.

For example:

File 1:

```
>>> set(element.font for element in document.elements)
{'EAAAAA+FreeMono,12.0', 'BAAAAA+LiberationSerif-Bold,16.0', 'CAAAAA+LiberationSerif,
↪12.0', 'DAAAAA+FreeMonoBold,12.0', 'BAAAAA+LiberationSerif-Bold,12.0'}
```

File 2:

```
>>> set(element.font for element in document.elements)
{'CIPKDS+FreeMono,12.0', 'FDHZTR+LiberationSerif-Bold,16.0', 'KJVFSL+LiberationSerif,
↪12.0', 'BXNKHF+FreeMonoBold,12.0', 'OKSDFT+LiberationSerif-Bold,12.0'}
```

In this case mapping fonts with regex patterns makes more sense. Create the your font mapping like before but fill it with regex patterns that don't specify the prefix precisely. Also specify that the font mapping contains regex patterns when loading the document.

```
FONT_MAPPING = {
    r"\w{6}\+LiberationSerif-Bold,16.0": "title",
    r"\w{6}\+LiberationSerif-Bold,12.0": "sub_title",
    r"\w{6}\+LiberationSerif,12.0": "text",
    r"\w{6}\+FreeMonoBold,12.0": "table_header",
    r"\w{6}\+FreeMono,12.0": "table_text",
}
document = load_file("order_summary.pdf", font_mapping=FONT_MAPPING, font_mapping_is_
↪regex=True)
```

2.2.4 Step 4 - Add sections

Another thing we can do to make our job easier is to add *Sections* to our document. A *Sections* class is made available on *document.sectioning*, which in particular allows us to call *create_section()*.

A section has a name, and contains all elements between the start element and the end element. You can add multiple sections with the same name, but each section will have both a `name` and a `unique_name` (which is just the name with an additional `_n` on the end, where `n` is the number of sections with that name).

As with the *PDFDocument*, a *Section* has an *elements* property which returns an *ElementList*, allowing you to filter the elements.

Important: Never instantiate a *Sections* yourself. You should always use *create_section()*.

Calling *create_section()* will return the *Section*, but the *Sectioning* class also has *get_section()* and *get_sections_with_name()* methods.

Going back to our example, we will create sections for the order summary table, and for the totals table. Our order summary table will start with the "Order Summary:" sub title and end at the "Totals:" sub title. Note that there are two elements on the page with text equal to "Order Summary:", however they have different font and so we can still extract exactly the one we want.

Order Summary:

Item	Unit Cost	Quantity	Cost
Challenger 100g Whole Hops	£3.29	1	£3.29
Maris Otter Pale Ale Malt (Crushed)	£1.50/1000g	4000g	£6.00
WLP037 Yorkshire Ale Yeast	£7.08	1	£7.08
Bottle Caps	£1 per 100	500	£5

Totals:

Subtotal:	£26.28
Shipping	£6
VAT 20%	£6.45
Total:	**£38.73**

By default, *create_section()* will include the last element in the section, but this can be disabled by passing `include_last_element=False`.

The totals section will run from the "Totals:" sub title, until the end of the document. An *ElementList* (e.g. `document.elements`) acts like a set of elements, but it does also define an order, and as such we can access the last element in the *ElementList* by simply doing `document.elements[-1]`.

```
order_summary_sub_title_element = (
    document.elements.filter_by_font("sub_title")
    .filter_by_text_equal("Order Summary:")
    .extract_single_element()
)

totals_sub_title_element = (
    document.elements.filter_by_font("sub_title")
    .filter_by_text_equal("Totals:")
    .extract_single_element()
)

final_element = document.elements[-1]

order_summary_section = document.sectioning.create_section(
    name="order_summary",
    start_element=order_summary_sub_title_element,
    end_element=totals_sub_title_element,
    include_last_element=False,
)
```

Again, the visualise tool is helpful to check everything worked as expected, as it will draw a border around all of our sections:

2.2.5 Step 5 - Extract tables

Now we have mapped our fonts and added some sections, we'd like to extract the table. In this case, we are able to use *extract_simple_table()*. We need to pass this the elements which form our table, however currently our sections also include the sub titles, "Order Summary:" and "Totals:". We need to exclude these from the elements we pass to *extract_simple_table()*. We have a reference to the sub title elements, so we could simply use *remove_element()*. However, since the tables seem to have their own fonts, it may be more robust to use *filter_by_fonts()*.

We will also pass `as_text=True`, since we are interested in the text, not the *PDFElements* themselves.

```
order_summary_table = tables.extract_simple_table(
    order_summary_section.elements.filter_by_fonts("table_header", "table_text"),
    as_text=True,
)

totals_table = tables.extract_simple_table(
    totals_section.elements.filter_by_fonts("table_header", "table_text"), as_
↪text=True
)
```

This gives:

```
>>> order_summary_table
[['Item', 'Unit Cost', 'Quantity', 'Cost'], ['Challenger 100g\nWhole Hops', '£3.29',
↪'1', '£3.29'], ['Maris Otter \nPale Ale Malt \n(Crushed)', '£1.50/1000g', '4000g',
↪'£6.00'], ['WLP037 \nYorkshire Ale \nYeast', '£7.08', '1', '£7.08'], ['Bottle Caps',
↪ '£1 per 100', '500', '£5']]

>>> totals_table
[['Subtotal:', '£26.28'], ['Shipping', '£6'], ['VAT 20%', '£6.45'], ['Total:', '£38.73
↪']]
```

As one final step, since the order summary table has a header row, we can make use of *add_header_to_table()*, which will change the list of lists to a list of dicts, mapping the header to the values in each row:

```
order_summary_with_header = tables.add_header_to_table(order_summary_table)
```

```
>>> order_summary_with_header
[{'Item': 'Challenger 100g\nWhole Hops', 'Unit Cost': '£3.29', 'Quantity': '1', 'Cost
↪': '£3.29'}, {'Item': 'Maris Otter \nPale Ale Malt \n(Crushed)', 'Unit Cost': '£1.
↪50/1000g', 'Quantity': '4000g', 'Cost': '£6.00'}, {'Item': 'WLP037 \nYorkshire Ale␣
↪\nYeast', 'Unit Cost': '£7.08', 'Quantity': '1', 'Cost': '£7.08'}, {'Item': 'Bottle␣
↪Caps', 'Unit Cost': '£1 per 100', 'Quantity': '500', 'Cost': '£5'}]
```

2.2.6 Full Code

```
from py_pdf_parser.loaders import load_file
from py_pdf_parser import tables

# from py_pdf_parser.visualise import visualise


# Step 1 - Load the file
document = load_file("order_summary.pdf")

# visualise(document)

# Step 2 - Use a font mapping

# Show all fonts:
# set(element.font for element in document.elements)

FONT_MAPPING = {
    "BAAAAA+LiberationSerif-Bold,16.0": "title",
    "BAAAAA+LiberationSerif-Bold,12.0": "sub_title",
    "CAAAAA+LiberationSerif,12.0": "text",
    "DAAAAA+FreeMonoBold,12.0": "table_header",
    "EAAAAA+FreeMono,12.0": "table_text",
}
document = load_file("order_summary.pdf", font_mapping=FONT_MAPPING)

# OR

# use regex patterns

FONT_MAPPING = {
    r"\w{6}\+LiberationSerif-Bold,16.0": "title",
    r"\w{6}\+LiberationSerif-Bold,12.0": "sub_title",
    r"\w{6}\+LiberationSerif,12.0": "text",
    r"\w{6}\+FreeMonoBold,12.0": "table_header",
    r"\w{6}\+FreeMono,12.0": "table_text",
}
document = load_file("order_summary.pdf", font_mapping=FONT_MAPPING, font_mapping_is_
↪regex=True)

# visualise(document)

# Step 3 - Add sections
order_summary_sub_title_element = (
    document.elements.filter_by_font("sub_title")
```

(continues on next page)

(continued from previous page)

```
    .filter_by_text_equal("Order Summary:")
    .extract_single_element()
)

totals_sub_title_element = (
    document.elements.filter_by_font("sub_title")
    .filter_by_text_equal("Totals:")
    .extract_single_element()
)

final_element = document.elements[-1]

order_summary_section = document.sectioning.create_section(
    name="order_summary",
    start_element=order_summary_sub_title_element,
    end_element=totals_sub_title_element,
    include_last_element=False,
)

totals_section = document.sectioning.create_section(
    name="totals", start_element=totals_sub_title_element, end_element=final_element
)

# visualise(document)

# Step 4 - Extract tables

order_summary_table = tables.extract_simple_table(
    order_summary_section.elements.filter_by_fonts("table_header", "table_text"),
    as_text=True,
)

totals_table = tables.extract_simple_table(
    totals_section.elements.filter_by_fonts("table_header", "table_text"), as_
↪text=True
)

order_summary_with_header = tables.add_header_to_table(order_summary_table)
```

2.3 More Tables

In this example, we will learn how to extract different types of table, and the difference between a simple table and more complicated ones.

You can `download the example here`.

Please read the *Order Summary* example first, as this covers some other functionality of the table extraction methods.

2.3.1 Load the file

The following code (click "show code" below to see it) loads the file, and assigns the elements for each table to a variable. If this does not make sense, you should go back and look at some of the previous examples.

```
from py_pdf_parser.loaders import load_file

FONT_MAPPING = {
    "BAAAAA+LiberationSerif-Bold,12.0": "header",
    "CAAAAA+LiberationSerif,12.0": "table_element",
}
document = load_file("tables.pdf", font_mapping=FONT_MAPPING)

headers = document.elements.filter_by_font("header")

# Extract reference elements
simple_table_header = headers.filter_by_text_equal(
    "Simple Table"
).extract_single_element()

simple_table_with_gaps_header = headers.filter_by_text_equal(
    "Simple Table with gaps"
).extract_single_element()

simple_table_with_gaps_in_first_row_col_header = headers.filter_by_text_equal(
    "Simple Table with gaps in first row/col"
).extract_single_element()

non_simple_table_header = headers.filter_by_text_equal(
    "Non Simple Table"
).extract_single_element()

non_simple_table_with_merged_cols_header = headers.filter_by_text_equal(
    "Non Simple Table with Merged Columns"
).extract_single_element()

non_simple_table_with_merged_rows_header = headers.filter_by_text_equal(
    "Non Simple Table with Merged Rows and Columns"
).extract_single_element()

over_the_page_header = headers.filter_by_text_equal(
    "Over the page"
).extract_single_element()

# Extract table elements
simple_table_elements = document.elements.between(
    simple_table_header, simple_table_with_gaps_header
)
simple_table_with_gaps_elements = document.elements.between(
    simple_table_with_gaps_header, simple_table_with_gaps_in_first_row_col_header
)

simple_table_with_gaps_in_first_row_col_elements = document.elements.between(
    simple_table_with_gaps_in_first_row_col_header, non_simple_table_header
)

non_simple_table_elements = document.elements.between(
    non_simple_table_header, non_simple_table_with_merged_cols_header
)

non_simple_table_with_merged_cols_elements = document.elements.between(
    non_simple_table_with_merged_cols_header, non_simple_table_with_merged_rows_header
```

(continues on next page)

(continued from previous page)

```
)

non_simple_table_with_merged_rows_and_cols_elements = document.elements.between(
    non_simple_table_with_merged_rows_header, over_the_page_header
)

over_the_page_elements = document.elements.after(over_the_page_header)
```

2.3.2 Overview

The tables in the example pdf are split into "Simple Tables" and "Non Simple Tables". For the simple tables, we will be able to use *extract_simple_table()*, otherwise we must use *extract_table()*. The former is much more efficient, and should be used when possible.

In general, tables can become more complicated by having missing cells, or merged cells which go across multiple columns or multiple rows. In both cases, you will have to pass additional parameters to stop exceptions being raised when this is the case. This is to make the extraction more robust, and protect against unexpected outcomes.

To use *extract_simple_table()* we must have at least one column and one row which have no missing cells, and we must have no merged cells at all. We will need to know which row/column has no missing cells, as these must be passed as the reference row and column.

To understand why: for each column element in the reference row and each row element in the reference column, *extract_simple_table()* will scan across from the row element (to get the row) and up/down from the column element (to get the column), and see if there is an element there. If there is, it is added to the table. Therefore, if there are gaps in the reference row/column, other elements may get missed. There is a check for this, so an exception will be raised if this is the case.

This means *extract_simple_table()* takes time proportional to `len(cols) + len(rows)`. Conversely, *extract_table()* is at least `len(cols) * len(rows)`, and if there are merged cells it will be even worse. (Note in reality the complexity is not quite this simple, but it should give you an idea of the difference.)

Below, we will work through increasingly complex examples to explain the functionality, and the steps involved.

2.3.3 Simple Table

This table is as simple as they come - there are no blank or merged cells. This means we can simply use *extract_simple_table()* as we have seen previously.

```
from py_pdf_parser import tables
table = tables.extract_simple_table(simple_table_elements, as_text=True)
```

```
>>> table
[['Heading 1', 'Heading 2', 'Heading 3', 'Heading 4'], ['A', '1', 'A', '1'], ['B', '2
↪', 'B', '2'], ['C', '3', 'C', '3']]
```

2.3.4 Simple Table with gaps

This table has gaps, however there are no gaps in the first row or column. These are the default reference row and column, and so *extract_simple_table()* will still work as expected. Blank cells will be empty strings if `as_text=True`, and otherwise they will be `None`. However, if we try the same code as above:

```
table = tables.extract_simple_table(
    simple_table_with_gaps_elements, as_text=True
)
```

this will raise an exception:

```
py_pdf_parser.exceptions.TableExtractionError: Element not found, there appears to be␣
↪a gap in the table. If this is expected, pass allow_gaps=True.
```

This is to allow py-pdf-parser to be more robust in the case that you're expecting your table to have no empty cells. As the error message says, since this is expected behaviour we can simply pass `allow_gaps=True`.

```
table = tables.extract_simple_table(
    simple_table_with_gaps_elements, as_text=True, allow_gaps=True
)
```

```
>>> table
[['Heading 1', 'Heading 2', 'Heading 3', 'Heading 4'], ['A', '1', '', '1'], ['B', '',
↪'', ''], ['C', '', 'C', '3']]
```

2.3.5 Simple Table with gaps in first row/col

This table is similar to the above example, but now we have gaps in the first row and the first column (if either of these were true then the above wouldn't work). If we try the above code, a useful exception is raised:

```
table = tables.extract_simple_table(
    simple_table_with_gaps_in_first_row_col_elements, as_text=True, allow_gaps=True
)
```

```
py_pdf_parser.exceptions.TableExtractionError: Number of elements in table (9) does␣
↪not match number of elements passed (12). Perhaps try extract_table instead of␣
↪extract_simple_table, or change you reference element.
```

The error message suggests either passing another reference element, or using the more complicated *`extract_table()`* method. In this case, as we still have a row and a column which have no missing cells, we can just pass a new reference element.

As such, we can use the second column and the last row as our references, as neither of these have missing cells. The reference row and column are specified by simply passing the unique element in both the reference row and the reference column (called the reference element). In this case, it's the first number "3" in the table. Here we will be lazy and simply use the fact that this is the 10th element in the table, but you should probably do something smarter.

```
reference_element = simple_table_with_gaps_in_first_row_col_elements[9]
table = tables.extract_simple_table(
    simple_table_with_gaps_in_first_row_col_elements,
    as_text=True,
    allow_gaps=True,
    reference_element=reference_element,
)
```

```
>>> table
[['Heading 1', 'Heading 2', '', 'Heading 4'], ['', '1', 'A', ''], ['B', '2', '', '2'],
↪ ['C', '3', 'C', '3']]
```

2.3.6 Non Simple Table

The next table does not have any row with no empty cells, and as such we must use *extract_table()*. There is no `allow_gaps` parameter for this method, since if you don't want to allow gaps you should be using *extract_simple_table()* instead.

Whilst the below may seem easier than working out the reference element in the above example, please note that it will be computationally slower.

```
table = tables.extract_table(non_simple_table_elements, as_text=True)
```

```
>>> table
[['', 'Heading 2', 'Heading 3', 'Heading 4'], ['A', '1', '', '1'], ['B', '', 'B', '2
↪'], ['C', '3', 'C', '']]
```

2.3.7 Non Simple Table with Merged Columns

This table has text which goes across multiple columns. If we naively run this as above:

```
table = tables.extract_table(non_simple_table_with_merged_cols_elements, as_text=True)
```

then we get an exception:

```
py_pdf_parser.exceptions.TableExtractionError: An element is in multiple columns. If␣
↪this is expected, you can try passing fix_element_in_multiple_cols=True
```

Just like `allow_gaps`, this is so we can be more robust in the case that this is not expected. The error helpfully suggests to try passing `fix_element_in_multiple_cols=True`.

```
table = tables.extract_table(
    non_simple_table_with_merged_cols_elements,
    as_text=True,
    fix_element_in_multiple_cols=True,
)
```

```
>>> table
[['Heading 1', 'Heading 2', 'Heading 3', 'Heading 4'], ['A', '1', 'A', '1'], ['This␣
↪text spans across multiple columns', '', 'B', '2'], ['C', '3', 'C', '3']]
```

Note that the merged cell has been pushed into the left-most column. Likewise, if we had a cell that was merged across multiple rows, we could pass `fix_element_in_multiple_rows=True`, and it would be pushed into the top row.

2.3.8 Non Simple Table with Merged Rows and Columns

In this case we have both merged rows and merged columns. We can pass both `fix_element_in_multiple_rows=True` and `fix_element_in_multiple_cols=True`. The merged cell will be pushed into the left-most column and the top row.

```
table = tables.extract_table(
    non_simple_table_with_merged_rows_and_cols_elements,
    as_text=True,
    fix_element_in_multiple_rows=True,
```

(continues on next page)

(continued from previous page)

```
    fix_element_in_multiple_cols=True,
)
```

```
>>> table
[['Heading 1', 'Heading 2', 'Heading 3', 'Heading 4'], ['This text spans across␣
↪multiple rows and \nmultiple columns.', '', 'A', '1'], ['', '', 'B', '2'], ['C', '3
↪', 'C', '3']]
```

2.3.9 Over the page

The final table goes over the page break. This is not a problem, simply pass the elements within the table and the result should be correct.

If you had e.g. a footer that broke the table in two, simply ensure these elements are not included in the element list you pass to *extract_table()*, and again it should still work.

```
table = tables.extract_simple_table(over_the_page_elements, as_text=True)
```

```
>>> table
[['Heading 1', 'Heading 2', 'Heading 3', 'Heading 4'], ['A', '1', 'A', '1'], ['B', '2
↪', 'B', '2'], ['C', '3', 'C', '3']]
```

2.4 Element Ordering

In this example, we see how to specify a custom ordering for the elements.

For this we will use a simple pdf, which has a single element in each corner of the page. You can `download the example here`.

2.4.1 Default

The default element ordering is left to right, top to bottom.

```
from py_pdf_parser.loaders import load_file

file_path = "grid.pdf"

# Default - left to right, top to bottom
document = load_file(file_path)
print([element.text() for element in document.elements])
```

This results in

```
['Top Left', 'Top Right', 'Bottom Left', 'Bottom Right']
```

2.4.2 Presets

There are also preset orderings for `right to left, top to bottom`, `top to bottom, left to right`, and `top to bottom, right to left`. You can use these by importing the *ElementOrdering*

class from *py_pdf_parser.components* and passing these as the `element_ordering` argument to *PDFDocument*. Note that keyword arguments to *load()* and *load_file()* get passed through to the *PDFDocument*.

```
from py_pdf_parser.loaders import load_file
from py_pdf_parser.components import ElementOrdering

# Preset - right to left, top to bottom
document = load_file(
    file_path, element_ordering=ElementOrdering.RIGHT_TO_LEFT_TOP_TO_BOTTOM
)
print([element.text() for element in document.elements])

# Preset - top to bottom, left to right
document = load_file(
    file_path, element_ordering=ElementOrdering.TOP_TO_BOTTOM_LEFT_TO_RIGHT
)
print([element.text() for element in document.elements])

# Preset - top to bottom, right to left
document = load_file(
    file_path, element_ordering=ElementOrdering.TOP_TO_BOTTOM_RIGHT_TO_LEFT
)
print([element.text() for element in document.elements])
```

which results in

```
['Top Right', 'Top Left', 'Bottom Right', 'Bottom Left']
['Bottom Left', 'Top Left', 'Bottom Right', 'Top Right']
['Top Right', 'Bottom Right', 'Top Left', 'Bottom Left']
```

2.4.3 Custom Ordering

If none of the presets give an ordering you are looking for, you can also pass a callable as the `element_ordering` argument of *PDFDocument*. This callable will be given a list of elements for each page, and should return a list of the same elements, in the desired order.

Important: The elements which get passed to your function will be PDFMiner.six elements, and NOT class `PDFElement`. You can access the `x0`, `x1`, `y0`, `y1` directly, and extract the text using *get_text()*. Other options are available: please familiarise yourself with the PDFMiner.six documentation.

Note: Your function will be called multiple times, once for each page of the document. Elements will always be considered in order of increasing page number, your function only controls the ordering within each page.

For example, if we wanted to implement an ordering which is bottom to top, left to right then we can do this as follows:

```
from py_pdf_parser.loaders import load_file

# Custom - bottom to top, left to right
def ordering_function(elements):
    """
    Note: Elements will be PDFMiner.six elements. The x axis is positive as you go␣
↪left
```

(continues on next page)

(continued from previous page)

```
    to right, and the y axis is positive as you go bottom to top, and hence we can
    simply sort according to this.
    """
    return sorted(elements, key=lambda elem: (elem.x0, elem.y0))


document = load_file(file_path, element_ordering=ordering_function)
print([element.text() for element in document.elements])
```

which results in

```
['Bottom Left', 'Top Left', 'Bottom Right', 'Top Right']
```

2.4.4 Multiple Columns

Finally, suppose our PDF has multiple columns, like `this example`.

If we don't specify an `element_ordering`, the elements will be extracted in the following order:

```
['Column 1 Title', 'Column 2 Title', 'Here is some column 1 text.', 'Here is some␣
↪column 2 text.', 'Col 1 left', 'Col 1 right', 'Col 2 left', 'Col 2 right']
```

If we visualise this document (see the *Simple Memo* example if you don't know how to do this), then we can see that the column divider is at an `x` value of about 300. Using this information, we can specify a custom ordering function which will order the elements left to right, top to bottom, but in each column individually.

```
from py_pdf_parser.loaders import load_file

document = load_file("columns.pdf")

def column_ordering_function(elements):
    """
    The first entry in the key is False for colum 1, and Tru for column 2. The second
    and third keys just give left to right, top to bottom.
    """
    return sorted(elements, key=lambda elem: (elem.x0 > 300, -elem.y0, elem.x0))


document = load_file(file_path, element_ordering=column_ordering_function)
print([element.text() for element in document.elements])
```

which returns the elements in the correct order:

```
['Column 1 Title', 'Here is some column 1 text.', 'Col 1 left', 'Col 1 right',
↪'Column 2 Title', 'Here is some column 2 text.', 'Col 2 left', 'Col 2 right']
```

2.5 Extracting Text From Figures

PDFs are structured documents, and can contain Figures. By default, PDFMiner.six and hence py-pdf-parser does not extract text from figures.

You can `download an example here`. In the example, there is figure which contains a red square, and some text. Below the figure there is some more text.

By default, the text in the figure will not be included:

```
from py_pdf_parser.loaders import load_file
document = load_file("figure.pdf")
print([element.text() for element in document.elements])
```

which results in:

```
["Here is some text outside of an image"]
```

To include the text inside the figure, we must pass the `all_texts` layout parameter. This is documented in the PDFMiner.six documentation, here.

The layout parameters can be passed to both *`load()`* and `load_file()` as a dictionary to the `la_params` argument.

In our case:

```
from py_pdf_parser.loaders import load_file
document = load_file("figure.pdf", la_params={"all_texts": True})
print([element.text() for element in document.elements])
```

which results in:

```
["This is some text in an image", "Here is some text outside of an image"]
```

CHAPTER 3

Reference

3.1 Common

class py_pdf_parser.common.**BoundingBox**(*x0: float*, *x1: float*, *y0: float*, *y1: float*)

A rectangle, stored using the coordinates (x0, y0) of the bottom left corner, and the coordinates (x1, y1) of the top right corner.

Parameters

- **x0** (*int*) – The x coordinate of the bottom left corner.
- **x1** (*int*) – The x coordinate of the top right corner.
- **y0** (*int*) – The y coordinate of the bottom left corner.
- **y1** (*int*) – The y coordinate of the top right corner.

Raises InvalidCoordinatesError – if x1 is smaller than x0 or y1 is smaller than y0.

x0

The x coordinate of the bottom left corner.

Type int

x1

The x coordinate of the top right corner.

Type int

y0

The y coordinate of the bottom left corner.

Type int

y1

The y coordinate of the top right corner.

Type int

width
The width of the box, equal to x1 - x0.

Type int

height
The height of the box, equal to y1 - y0.

Type int

3.2 Components

class py_pdf_parser.components.**ElementOrdering**
A class enumerating the available presets for element_ordering.

class py_pdf_parser.components.**PDFDocument**(*pages: Dict[int, Page], pdf_file_path: Optional[str] = None, font_mapping: Optional[Dict[str, str]] = None, font_mapping_is_regex: bool = False, regex_flags: Union[int, re.RegexFlag] = 0, font_size_precision: int = 1, element_ordering: Union[py_pdf_parser.components.ElementOrdering, Callable[[List[T]], List[T]]] = <ElementOrdering.LEFT_TO_RIGHT_TOP_TO_BOTTOM: 1>*)
Contains all information about the whole pdf document.

To instantiate, you should pass a dictionary mapping page numbers to pages, where each page is a Page named-tuple containing the width and heigh of the page, and a list of pdf elements (which should be directly from PDFMiner, i.e. should be PDFMiner *LTComponent's). On instantiation, the PDFDocument will convert all of these into 'PDFElement* classes.

Parameters

- **pages** (*dict[int, Page]*) – A dictionary mapping page numbers (int) to pages, where pages are a *Page* namedtuple (containing a width, height and a list of elements from PDFMiner).
- **pdf_file_path** (*str, optional*) – A file path to the PDF file. This is optional, and is only used to display your pdf as a background image when using the visualise functions.
- **font_mapping** (*dict, optional*) – *PDFElement's have a 'font* attribute, and the font is taken from the PDF. You can map these fonts to instead use your own internal font names by providing a font_mapping. This is a dictionary with keys being the original font (including font size) and values being your new names.
- **font_mapping_is_regex** (*bool, optional*) – Indicates whether font_mapping keys should be considered as regexes. In this case all the fonts will be matched with the regexes. It is only relevant if font_mapping is not None. Default: False.
- **regex_flags** (*str, optional*) – Regex flags compatible with the re module. Default: 0.
- **font_size_precision** (*int*) – How much rounding to apply to the font size. The font size will be rounded to this many decimal places.

- **element_ordering** (ElementOrdering *or callable, optional*) – An ordering function for the elements. Either a member of the ElementOrdering Enum, or a callable which takes a list of elements and returns an ordered list of elements. This will be called separately for each page. Note that the elements in this case will be PDFMiner elements, and not PDFElements from this package.

number_of_pages
The total number of pages in the document.

Type int

page_numbers
A list of available page numbers.

Type list(int)

sectioning
Gives access to the sectioning utilities. See the documentation for the *Sectioning* class.

elements
An ElementList containing all elements in the document.

Returns All elements in the document.

Return type *ElementList*

fonts
A set of all the fonts in the document.

Returns All the fonts in the document.

Return type set[str]

get_page(*page_number: int*) → py_pdf_parser.components.PDFPage
Returns the *PDFPage* for the specified *page_number*.

Parameters **page_number** (*int*) – The page number.

Raises PageNotFoundError – If *page_number* was not found.

Returns The requested page.

Return type *PDFPage*

pages
A list of all pages in the document.

Returns All pages in the document.

Return type list[*PDFPage*]

class py_pdf_parser.components.**PDFElement**(*document: PDFDocument*, *element: LTComponent*, *index: int*, *page_number: int*, *font_size_precision: int = 1*)
A representation of a single element within the pdf.

You should not instantiate this yourself, but should let the *PDFDocument* do this.

Parameters

- **document** (PDFDocument) – A reference to the *PDFDocument*.
- **element** (*LTComponent*) – A PDF Miner LTComponent.
- **index** (*int*) – The index of the element within the document.
- **page_number** (*int*) – The page number that the element is on.

- **font_size_precision** (*int*) – How much rounding to apply to the font size. The font size will be rounded to this many decimal places.

original_element
A reference to the original PDF Miner element.

Type LTComponent

tags
A list of tags that have been added to the element.

Type set[str]

bounding_box
The box representing the location of the element.

Type *BoundingBox*

add_tag(*new_tag: str*) → None
Adds the *new_tag* to the tags set.

Parameters **new_tag** (*str*) – The tag you would like to add.

entirely_within(*bounding_box: py_pdf_parser.common.BoundingBox*) → bool
Whether the entire element is within the bounding box.

Parameters **bounding_box** (BoundingBox) – The bounding box to check whether the element is within.

Returns True if the element is entirely contained within the bounding box.

Return type bool

font
The name and size of the font, separated by a comma with no spaces.

This will be taken from the pdf itself, using the first character in the element.

If you have provided a font_mapping, this is the string you should map. If the string is mapped in your font_mapping then the mapped value will be returned. font_mapping can have regexes as keys.

Returns The font of the element.

Return type str

font_name
The name of the font.

This will be taken from the pdf itself, using the most common font within all the characters in the element.

Returns The font name of the element.

Return type str

font_size
The size of the font.

This will be taken from the pdf itself, using the most common size within all the characters in the element.

Returns

The font size of the element, rounded to the font_size_precision of the document.

Return type float

ignore() → None
Marks the element as ignored.

The element will no longer be returned in any newly instantiated *ElementList*. Note that this includes calling any new filter functions on an existing *ElementList*, since doing so always returns a new *ElementList*.

ignored
A flag specifying whether the element has been ignored.

page_number
The page_number of the element in the document.

Returns The page number of the element.

Return type int

partially_within(*bounding_box: py_pdf_parser.common.BoundingBox*) → bool
Whether any part of the element is within the bounding box.

Parameters **bounding_box** (BoundingBox) – The bounding box to check whether the element is partially within.

Returns True if any part of the element is within the bounding box.

Return type bool

text(*stripped: bool = True*) → str
The text contained in the element.

Parameters **stripped** (*bool, optional*) – Whether to strip the text of the element. Default: True.

Returns The text contained in the element.

Return type str

class py_pdf_parser.components.**PDFPage**(*document: py_pdf_parser.components.PDFDocument*, *width: int*, *height: int*, *page_number: int*, *start_element: py_pdf_parser.components.PDFElement*, *end_element: py_pdf_parser.components.PDFElement*)
A representation of a page within the *PDFDocument*.

We store the width, height and page number of the page, along with the first and last element on the page. Because the elements are ordered, this allows us to easily determine all the elements on the page.

Parameters

- **document** (PDFDocument) – A reference to the *PDFDocument*.
- **width** (*int*) – The width of the page.
- **height** (*int*) – The height of the page.
- **page_number** (*int*) – The page number.
- **start_element** (PDFElement) – The first element on the page.
- **end_element** (PDFElement) – The last element on the page.

elements
Returns an *ElementList* containing all elements on the page.

Returns All the elements on the page.

Return type *ElementList*

3.3 Filtering

class py_pdf_parser.filtering.**ElementList**(*document: PDFDocument, indexes: Union[Set[int], FrozenSet[int], None] = None*)

Used to represent a list of elements, and to enable filtering of those elements.

Any time you have a group of elements, for example pdf_document.elements or page.elements, you will get an *ElementList*. You can iterate through this, and also access specific elements. On top of this, there are lots of methods which you can use to further filter your elements. Since all of these methods return a new ElementList, you can chain these operations.

Internally, we keep a set of indexes corresponding to the PDFElements in the document. This means you can treat ElementLists like sets to combine different ElementLists together.

We often implement pluralised versions of methods, which is a shortcut to applying the or operator | to multiple ElementLists with the singular version applied, for example *foo.filter_by_tags("bar", "baz")* is the same as *foo.filter_by_tag("bar") | foo.filter_by_tag("baz")*.

Similarly, chaining two filter commands is the same as applying the & operator, for example *foo.filter_by_tag("bar").filter_by_tag("baz")* is the same as *foo.filter_by_tag("bar") & foo.filter_by_tag("baz")*. Note that this is not the case for methods which do not filter, e.g. *add_element*.

Ignored elements will be excluded on instantiation. Each time you chain a new filter a new ElementList is returned. Note this will remove newly-ignored elements.

Note: As ElementList is implemented using sets internally, you will not be able to have an element in an ElementList multiple times.

Parameters

- **document** (PDFDocument) – A reference to the PDF document
- **indexes** (*set, optional*) – A set (or frozenset) of element indexes. Defaults to all elements in the document.

document
A reference to the PDF document.

Type *PDFDocument*

indexes
A frozenset of element indexes.

Type set, optional

__and__(*other: py_pdf_parser.filtering.ElementList*) → py_pdf_parser.filtering.ElementList
Returns an ElementList of elements that are in both ElementList

__contains__(*element: PDFElement*) → bool
Returns True if the element is in the ElementList, otherwise False.

__eq__(*other: object*) → bool
Returns True if the two ElementLists contain the same elements from the same document.

__getitem__(*key: Union[int, slice]*) → Union[PDFElement, ElementList]
Returns the element in position *key* of the ElementList if an int is given, or returns a new ElementList if a slice is given.

Elements are ordered by their original positions in the document, which is left-to-right, top-to-bottom (the same you you read).

__hash__() → int
Return hash(self).

__init__(*document: PDFDocument, indexes: Union[Set[int], FrozenSet[int], None] = None*)
Initialize self. See help(type(self)) for accurate signature.

__iter__() → py_pdf_parser.filtering.ElementIterator
Returns an ElementIterator class that allows iterating through elements.

Elements will be returned in order of the elements in the document, left-to-right, top-to-bottom (the same as you read).

__len__() → int
Returns the number of elements in the ElementList.

__or__(*other: py_pdf_parser.filtering.ElementList*) → py_pdf_parser.filtering.ElementList
Returns an ElementList of elements that are in either ElementList

__repr__() → str
Return repr(self).

__sub__(*other: py_pdf_parser.filtering.ElementList*) → py_pdf_parser.filtering.ElementList
Returns an ElementList of elements that are in the first ElementList but not in the second.

__weakref__
list of weak references to the object (if defined)

__xor__(*other: py_pdf_parser.filtering.ElementList*) → py_pdf_parser.filtering.ElementList
Returns an ElementList of elements that are in either ElementList, but not both.

above(*element: PDFElement*, *inclusive: bool = False*, *all_pages: bool = False*, *tolerance: float = 0.0*)
→ ElementList
Returns all elements which are above the given element.

If you draw a box from the bottom edge of the element to the bottom of the page, all elements which are partially within this box are returned. By default, only elements on the same page as the given element are included, but you can pass *inclusive=True* to also include the pages which come before (and so are above) the page containing the given element.

Note: By "above" we really mean "directly above", i.e. the returned elements all have at least some part which is horizontally aligned with the specified element.

Note: Technically the element you specify will satisfy the condition, but we assume you do not want that element returned. If you do, you can pass *inclusive=True*.

Parameters

- **element** (PDFElement) – The element in question.
- **inclusive** (*bool, optional*) – Whether the include *element* in the returned results. Default: False.
- **all_pages** (*bool, optional*) – Whether to included pages other than the page which the element is on.

- **tolerance** (*int, optional*) – To be counted as above, the elements must overlap by at least *tolerance* on the X axis. Tolerance is capped at half the width of the element. Default 0.

Returns The filtered list.

Return type *ElementList*

add_element (*element: PDFElement*) → ElementList
Explicitly adds the element to the ElementList.

Note: If the element is already in the ElementList, this does nothing.

Parameters **element** (PDFElement) – The element to add.

Returns A new list with the additional element.

Return type *ElementList*

add_elements (**elements*) → ElementList
Explicitly adds the elements to the ElementList.

Note: If the elements is already in the ElementList, this does nothing.

Parameters ***elements** (PDFElement) – The elements to add.

Returns A new list with the additional elements.

Return type *ElementList*

add_tag_to_elements (*tag: str*) → None
Adds a tag to all elements in the list.

Parameters **tag** (*str*) – The tag you would like to add.

after (*element: PDFElement*, *inclusive: bool = False*) → ElementList
Returns all elements after the specified element.

By after, we mean succeeding elements according to their index. The PDFDocument will order elements according to the specified element_ordering (which defaults to left to right, top to bottom).

Parameters

- **element** (PDFElement) – The element in question.
- **inclusive** (*bool, optional*) – Whether the include *element* in the returned results. Default: False.

Returns The filtered list.

Return type *ElementList*

before (*element: PDFElement*, *inclusive: bool = False*) → ElementList
Returns all elements before the specified element.

By before, we mean preceding elements according to their index. The PDFDocument will order elements according to the specified element_ordering (which defaults to left to right, top to bottom).

Parameters

- **element** (PDFElement) – The element in question.
- **inclusive** (*bool, optional*) – Whether the include *element* in the returned results. Default: False.

Returns The filtered list.

Return type *ElementList*

below(*element: PDFElement*, *inclusive: bool = False*, *all_pages: bool = False*, *tolerance: float = 0.0*) → ElementList

Returns all elements which are below the given element.

If you draw a box from the bottom edge of the element to the bottom of the page, all elements which are partially within this box are returned. By default, only elements on the same page as the given element are included, but you can pass *inclusive=True* to also include the pages which come after (and so are below) the page containing the given element.

Note: By "below" we really mean "directly below", i.e. the returned elements all have at least some part which is horizontally aligned with the specified element.

Note: Technically the element you specify will satisfy the condition, but we assume you do not want that element returned. If you do, you can pass *inclusive=True*.

Parameters

- **element** (PDFElement) – The element in question.
- **inclusive** (*bool, optional*) – Whether the include *element* in the returned results. Default: False.
- **all_pages** (*bool, optional*) – Whether to included pages other than the page which the element is on.
- **tolerance** (*int, optional*) – To be counted as below, the elements must overlap by at least *tolerance* on the X axis. Tolerance is capped at half the width of the element. Default 0.

Returns The filtered list.

Return type *ElementList*

between(*start_element: PDFElement*, *end_element: PDFElement*, *inclusive: bool = False*) → ElementList

Returns all elements between the start and end elements.

This is done according to the element indexes. The PDFDocument will order elements according to the specified element_ordering (which defaults to left to right, top to bottom).

This is the same as applying *before* with *start_element* and *after* with *end_element*.

Parameters

- **start_element** (PDFElement) – Returned elements will be after this element.
- **end_element** (PDFElement) – Returned elements will be before this element.
- **inclusive** (*bool, optional*) – Whether the include *start_element* and *end_element* in the returned results. Default: False.

Returns The filtered list.

Return type *ElementList*

extract_single_element() → PDFElement
Returns only element in the ElementList, provided there is only one element.

This is mainly for convenience, when you think you've filtered down to a single element and you would like to extract said element.

Raises

- NoElementFoundError – If there are no elements in the ElementList
- MultipleElementsFoundError – If there is more than one element in the ElementList

Returns The single element remaining in the list.

Return type *PDFElement*

filter_by_font(*font: str*) → py_pdf_parser.filtering.ElementList
Filter for elements containing only the given font.

Parameters **font** (*str*) – The font to filter for.

Returns The filtered list.

Return type *ElementList*

filter_by_fonts(**fonts*) → py_pdf_parser.filtering.ElementList
Filter for elements containing only the given font.

Parameters ***fonts** (*str*) – The fonts to filter for.

Returns The filtered list.

Return type *ElementList*

filter_by_page(*page_number: int*) → py_pdf_parser.filtering.ElementList
Filter for elements on the given page.

Parameters **page** (*int*) – The page to filter for.

Returns The filtered list.

Return type *ElementList*

filter_by_pages(**page_numbers*) → py_pdf_parser.filtering.ElementList
Filter for elements on any of the given pages.

Parameters ***pages** (*int*) – The pages to filter for.

Returns The filtered list.

Return type *ElementList*

filter_by_regex(*regex: str*, *regex_flags: Union[int, re.RegexFlag] = 0*, *stripped: bool = True*) → py_pdf_parser.filtering.ElementList
Filter for elements given a regular expression.

Parameters

- **regex** (*str*) – The regex to filter for.
- **regex_flags** (*str, optional*) – Regex flags compatible with the re module. Default: 0.
- **stripped** (*bool, optional*) – Whether to strip the text of the element before comparison. Default: True.

Returns The filtered list.

Return type *ElementList*

filter_by_section(*section_str: str*) → py_pdf_parser.filtering.ElementList
Filter for elements within the given section.

See the sectioning documentation for more details.

Parameters **section_name** (*str*) – The section to filter for.

Note: You need to specify an exact section, not just the name (i.e. "foo_0" not just "foo").

Returns The filtered list.

Return type *ElementList*

filter_by_section_name(*section_name: str*) → py_pdf_parser.filtering.ElementList
Filter for elements within any section with the given name.

See the sectioning documentation for more details.

Parameters **section_name** (*str*) – The section name to filter for.

Returns The filtered list.

Return type *ElementList*

filter_by_section_names(**section_names*) → py_pdf_parser.filtering.ElementList
Filter for elements within any section with any of the given names.

See the sectioning documentation for more details.

Parameters ***section_names** (*str*) – The section names to filter for.

Returns The filtered list.

Return type *ElementList*

filter_by_sections(**section_strs*) → py_pdf_parser.filtering.ElementList
Filter for elements within any of the given sections.

See the sectioning documentation for more details.

Parameters ***section_names** (*str*) – The sections to filter for.

Note: You need to specify an exact section, not just the name (i.e. "foo_0" not just "foo").

Returns The filtered list.

Return type *ElementList*

filter_by_tag(*tag: str*) → py_pdf_parser.filtering.ElementList
Filter for elements containing only the given tag.

Parameters **tag** (*str*) – The tag to filter for.

Returns The filtered list.

Return type *ElementList*

filter_by_tags(**tags*) → py_pdf_parser.filtering.ElementList
Filter for elements containing any of the given tags.

Parameters ***tags** (*str*) – The tags to filter for.

Returns The filtered list.

Return type *ElementList*

filter_by_text_contains(*text: str*) → py_pdf_parser.filtering.ElementList
Filter for elements whose text contains the given string.

Parameters **text** (*str*) – The text to filter for.

Returns The filtered list.

Return type *ElementList*

filter_by_text_equal(*text: str*, *stripped: bool = True*) → py_pdf_parser.filtering.ElementList
Filter for elements whose text is exactly the given string.

Parameters

- **text** (*str*) – The text to filter for.
- **stripped** (*bool, optional*) – Whether to strip the text of the element before comparison. Default: True.

Returns The filtered list.

Return type *ElementList*

filter_partially_within_bounding_box(*bounding_box: py_pdf_parser.common.BoundingBox*, *page_number: int*) → py_pdf_parser.filtering.ElementList
Returns all elements on the given page which are partially within the given box.

Parameters

- **bounding_box** (BoundingBox) – The bounding box to filter within.
- **page_number** (*int*) – The page which you'd like to filter within the box.

Returns The filtered list.

Return type *ElementList*

horizontally_in_line_with(*element: PDFElement*, *inclusive: bool = False*, *tolerance: float = 0.0*) → ElementList
Returns all elements which are horizontally in line with the given element.

If you extend the top and bottom edges of the element to the left and right of the page, all elements which are partially within this box are returned.

This is equivalent to doing *foo.to_the_left_of(...) | foo.to_the_right_of(...)*.

Note: Technically the element you specify will satisfy the condition, but we assume you do not want that element returned. If you do, you can pass *inclusive=True*.

Parameters

- **element** (PDFElement) – The element in question.
- **inclusive** (*bool, optional*) – Whether the include *element* in the returned results. Default: False.

- **tolerance** (*int, optional*) – To be counted as in line with, the elements must overlap by at least *tolerance* on the Y axis. Tolerance is capped at half the width of the element. Default 0.

Returns The filtered list.

Return type *ElementList*

ignore_elements() → None
Marks all the elements in the ElementList as ignored.

move_backwards_from(*element: PDFElement*, *count: int = 1*, *capped: bool = False*) → PDFElement
Returns the element in the element list obtained by moving backwards from *element* by *count*.

Parameters

- **element** (PDFElement) – The element to start at.
- **count** (*int, optional*) – How many elements to move from *element*. The default of 1 will move backwards by one element. Passing 0 will simply return the element itself. You can also pass negative integers to move forwards.
- **capped** (*bool, optional*) – By default (False), if the count is high enough that we try to move out of range of the list, an exception will be raised. Passing *capped=True* will change this behaviour to instead return the element at the start or end of the list.

Raises ElementOutOfRangeError – If the count is large (or large-negative) enough that we reach the start (or end) of the list. Only happens when capped=False.

move_forwards_from(*element: PDFElement*, *count: int = 1*, *capped: bool = False*) → PDFElement
Returns the element in the element list obtained by moving forwards from *element* by *count*.

Parameters

- **element** (PDFElement) – The element to start at.
- **count** (*int, optional*) – How many elements to move from *element*. The default of 1 will move forwards by one element. Passing 0 will simply return the element itself. You can also pass negative integers to move backwards.
- **capped** (*bool, optional*) – By default (False), if the count is high enough that we try to move out of range of the list, an exception will be raised. Passing *capped=True* will change this behaviour to instead return the element at the start or end of the list.

Raises ElementOutOfRangeError – If the count is large (or large-negative) enough that we reach the end (or start) of the list. Only happens when capped=False.

remove_element(*element: PDFElement*) → ElementList
Explicitly removes the element from the ElementList.

Note: If the element is not in the ElementList, this does nothing.

Parameters **element** (PDFElement) – The element to remove.

Returns A new list without the element.

Return type *ElementList*

remove_elements(**elements*) → ElementList
Explicitly removes the elements from the ElementList.

Note: If the elements are not in the ElementList, this does nothing.

Parameters ***elements** (PDFElement) – The elements to remove.

Returns A new list without the elements.

Return type *ElementList*

to_the_left_of(*element: PDFElement*, *inclusive: bool = False*, *tolerance: float = 0.0*) → ElementList
Filter for elements which are to the left of the given element.

If you draw a box from the left hand edge of the element to the left hand side of the page, all elements which are partially within this box are returned.

Note: By "to the left of" we really mean "directly to the left of", i.e. the returned elements all have at least some part which is vertically aligned with the specified element.

Note: Technically the element you specify will satisfy the condition, but we assume you do not want that element returned. If you do, you can pass *inclusive=True*.

Parameters

- **element** (PDFElement) – The element in question.
- **inclusive** (*bool, optional*) – Whether the include *element* in the returned results. Default: False.
- **tolerance** (*int, optional*) – To be counted as to the left, the elements must overlap by at least *tolerance* on the Y axis. Tolerance is capped at half the height of the element. Default 0.

Returns The filtered list.

Return type *ElementList*

to_the_right_of(*element: PDFElement*, *inclusive: bool = False*, *tolerance: float = 0.0*) → ElementList
Filter for elements which are to the right of the given element.

If you draw a box from the right hand edge of the element to the right hand side of the page, all elements which are partially within this box are returned.

Note: By "to the right of" we really mean "directly to the right of", i.e. the returned elements all have at least some part which is vertically aligned with the specified element.

Note: Technically the element you specify will satisfy the condition, but we assume you do not want that element returned. If you do, you can pass *inclusive=True*.

Parameters

- **element** (PDFElement) – The element in question.
- **inclusive** (*bool, optional*) – Whether the include *element* in the returned results. Default: False.
- **tolerance** (*int, optional*) – To be counted as to the right, the elements must overlap by at least *tolerance* on the Y axis. Tolerance is capped at half the height of the element. Default 0.

Returns The filtered list.

Return type *ElementList*

vertically_in_line_with(*element: PDFElement*, *inclusive: bool = False*, *all_pages: bool = False*, *tolerance: float = 0.0*) → ElementList

Returns all elements which are vertically in line with the given element.

If you extend the left and right edges of the element to the top and bottom of the page, all elements which are partially within this box are returned. By default, only elements on the same page as the given element are included, but you can pass *inclusive=True* to include all pages.

This is equivalent to doing *foo.above(...) | foo.below(...)*.

Note: Technically the element you specify will satisfy the condition, but we assume you do not want that element returned. If you do, you can pass *inclusive=True*.

Parameters

- **element** (PDFElement) – The element in question.
- **inclusive** (*bool, optional*) – Whether the include *element* in the returned results. Default: False.
- **all_pages** (*bool, optional*) – Whether to included pages other than the page which the element is on.
- **tolerance** (*int, optional*) – To be counted as in line with, the elements must overlap by at least *tolerance* on the X axis. Tolerance is capped at half the width of the element. Default 0.

Returns The filtered list.

Return type *ElementList*

3.4 Loaders

class py_pdf_parser.loaders.**Page**

This is used to pass PDF Miner elements of a page when instantiating PDFDocument.

Parameters

- **width** (*int*) – The width of the page.
- **height** (*int*) – The height of the page.
- **elements** (*list*) – A list of PDF Miner elements (LTTextBox) on the page.

elements
Alias for field number 2

height
Alias for field number 1

width
Alias for field number 0

py_pdf_parser.loaders.**load**(*pdf_file: IO*, *pdf_file_path: Optional[str] = None*, *la_params: Optional[Dict[KT, VT]] = None*, ***kwargs*) → py_pdf_parser.components.PDFDocument

Loads the pdf file into a PDFDocument.

Parameters

- **pdf_file** (*io*) – The PDF file.
- **la_params** (*dict*) – The layout parameters passed to PDF Miner for analysis. See the PDFMiner documentation here: https://pdfminersix.readthedocs.io/en/latest/reference/composable.html#laparams. Note that py_pdf_parser will re-order the elements it receives from PDFMiner so options relating to element ordering will have no effect.
- **pdf_file_path** (*str, optional*) – Passed to *PDFDocument*. See the documentation for *PDFDocument*.
- **kwargs** – Passed to *PDFDocument*. See the documentation for *PDFDocument*.

Returns A PDFDocument with the file loaded.

Return type *PDFDocument*

py_pdf_parser.loaders.**load_file**(*path_to_file: str*, *la_params: Optional[Dict[KT, VT]] = None*, ***kwargs*) → py_pdf_parser.components.PDFDocument

Loads a file according to the specified file path.

All other arguments are passed to *load*, see the documentation for *load*.

Returns A PDFDocument with the specified file loaded.

Return type *PDFDocument*

3.5 Sectioning

class py_pdf_parser.sectioning.**Section**(*document: PDFDocument*, *name: str*, *unique_name: str*, *start_element: PDFElement*, *end_element: PDFElement*)

A continuous group of elements within a document.

A section is intended to label a group of elements. Said elements must be continuous in the document.

> **Warning:** You should not instantiate a Section class yourself, but should call *create_section* from the *Sectioning* class below.

Parameters

- **document** (PDFDocument) – A reference to the document.
- **name** (*str*) – The name of the section.

- **unique_name** (*str*) – Multiple sections can have the same name, but a unique name will be generated by the Sectioning class.
- **start_element** (PDFElement) – The first element in the section.
- **end_element** (PDFElement) – The last element in the section.

elements
All the elements in the section.

Returns All the elements in the section.

Return type *ElementList*

class py_pdf_parser.sectioning.**Sectioning**(*document: PDFDocument*)
A sectioning utilities class, made available on all PDFDocuments as `.sectioning`.

create_section(*name: str*, *start_element: PDFElement*, *end_element: PDFElement*, *include_last_element: bool = True*) → Section
Creates a new section with the specified name.

Creates a new section with the specified name, starting at *start_element* and ending at *end_element* (inclusive). The unique name will be set to name_<idx> where <idx> is the number of existing sections with that name.

Parameters

- **name** (*str*) – The name of the new section.
- **start_element** (PDFElement) – The first element in the section.
- **end_element** (PDFElement) – The last element in the section.
- **include_last_element** (*bool*) – Whether the end_element should be included in the section, or only the elements which are strictly before the end element. Default: True (i.e. include end_element).

Returns The created section.

Return type *Section*

Raises InvalidSectionError – If a the created section would be invalid. This is usually because the end_element comes after the start element.

get_section(*unique_name: str*) → py_pdf_parser.sectioning.Section
Returns the section with the given unique name.

Raises SectionNotFoundError – If there is no section with the given unique_name.

get_sections_with_name(*name: str*) → Generator[py_pdf_parser.sectioning.Section, None, None]
Returns a list of all sections with the given name.

sections
Returns the list of all created Sections.

3.6 Tables

py_pdf_parser.tables.**add_header_to_table**(*table: List[List[str]], header: Optional[List[str]] = None*) → List[Dict[str, str]]
Given a table (list of lists) of strings, returns a list of dicts mapping the table header to the values.

Given a table, a list of rows which are lists of strings, returns a new table which is a list of rows which are dictionaries mapping the header values to the table values.

Parameters

- **table** – The table (a list of lists of strings).
- **header** (*list, optional*) – The header to use. If not provided, the first row of the table will be used instead. Your header must be the same width as your table, and cannot contain the same entry multiple times.

Raises InvalidTableHeaderError – If the width of the header does not match the width of the table, or if the header contains duplicate entries.

Returns A list of dictionaries, where each entry in the list is a row in the table, and a row in the table is represented as a dictionary mapping the header to the values.

Return type list[dict]

py_pdf_parser.tables.**extract_simple_table**(*elements: ElementList*, *as_text: bool = False*, *strip_text: bool = True*, *allow_gaps: bool = False*, *reference_element: Optional[PDFElement] = None*, *tolerance: float = 0.0*, *remove_duplicate_header_rows: bool = False*) → List[List[T]]

Returns elements structured as a table.

Given an ElementList, tries to extract a structured table by examining which elements are aligned.

To use this function, there must be at least one full row and one full column (which we call the reference row and column), i.e. the reference row must have an element in every column, and the reference column must have an element in every row. The reference row and column can be specified by passing the single element in both the reference row and the reference column. By default, this is the top left element, which means we use the first row and column as the references. Note if you need to change the reference_element, that means you have gaps in your table, and as such you will need to pass *allow_gaps=True*.

Important: This function uses the elements in the reference row and column to scan horizontally and vertically to find the rest of the table. If there are gaps in your reference row and column, this could result in rows and columns being missed by this function.

There must be a clear gap between each row and between each column which contains no elements, and a single cell cannot contain multiple elements.

If there are no valid reference rows or columns, try extract_table() instead. If you have elements spanning multiple rows or columns, it may be possible to fix this by using extract_table(). If you fail to satisfy any of the other conditions listed above, that case is not yet supported.

Parameters

- **elements** (ElementList) – A list of elements to extract into a table.
- **as_text** (*bool, optional*) – Whether to extract the text from each element instead of the PDFElement itself. Default: False.
- **strip_text** (*bool, optional*) – Whether to strip the text for each element of the table (Only relevant if as_text is True). Default: True.
- **allow_gaps** (*bool, optional*) – Whether to allow empty spaces in the table.
- **reference_element** (PDFElement, *optional*) – An element in a full row and a full column. Will be used to specify the reference row and column. If None, the top left element will be used, meaning the top row and left column will be used. If there are gaps in these, you should specify a different reference. Default: None.

- **tolerance** (*int, optional*) – For elements to be counted as in the same row or column, they must overlap by at least *tolerance*. Default: 0.
- **remove_duplicate_header_rows** (*bool, optional*) – Remove duplicates of the header row (the first row) if they exist. Default: False.

Raises TableExtractionError – If something goes wrong.

Returns

a list of rows, which are lists of PDFElements or strings (depending on the value of as_text).

Return type list[list]

py_pdf_parser.tables.**extract_table**(*elements: ElementList*, *as_text: bool = False*, *strip_text: bool = True*, *fix_element_in_multiple_rows: bool = False*, *fix_element_in_multiple_cols: bool = False*, *tolerance: float = 0.0*, *remove_duplicate_header_rows: bool = False*) → List[List[T]]

Returns elements structured as a table.

Given an ElementList, tries to extract a structured table by examining which elements are aligned. There must be a clear gap between each row and between each column which contains no elements, and a single cell cannot contain multiple elements.

If you fail to satisfy any of the other conditions listed above, that case is not yet supported.

Note: If you satisfy the conditions to use extract_simple_table, then that should be used instead, as it's much more efficient.

Parameters

- **elements** (ElementList) – A list of elements to extract into a table.
- **as_text** (*bool, optional*) – Whether to extract the text from each element instead of the PDFElement itself. Default: False.
- **strip_text** (*bool, optional*) – Whether to strip the text for each element of the table (Only relevant if as_text is True). Default: True.
- **fix_element_in_multiple_rows** (*bool, optional*) – If a table element is in line with elements in multiple rows, a TableExtractionError will be raised unless this argument is set to True. When True, any elements detected in multiple rows will be placed into the first row. This is only recommended if you expect this to be the case in your table. Default: False.
- **fix_element_in_multiple_cols** (*bool, optional*) – If a table element is in line with elements in multiple cols, a TableExtractionError will be raised unless this argument is set to True. When True, any elements detected in multiple cols will be placed into the first col. This is only recommended if you expect this to be the case in your table. Default: False.
- **tolerance** (*int, optional*) – For elements to be counted as in the same row or column, they must overlap by at least *tolerance*. Default: 0.
- **remove_duplicate_header_rows** (*bool, optional*) – Remove duplicates of the header row (the first row) if they exist. Default: False.

Raises TableExtractionError – If something goes wrong.

Returns

a list of rows, which are lists of PDFElements or strings (depending on the value of as_text).

Return type list[list]

py_pdf_parser.tables.**get_text_from_table**(*table: List[List[Optional[PDFElement]]]*, *strip_text: bool = True*) → List[List[str]]

Given a table (of PDFElements or None), returns a table (of element.text() or '').

Parameters

- **table** – The table (a list of lists of PDFElements).
- **strip_text** (*bool, optional*) – Whether to strip the text for each element of the table. Default: True.

Returns a list of rows, which are lists of strings.

Return type list[list[str]]

3.7 Visualise

py_pdf_parser.visualise.main.**visualise**(*document: py_pdf_parser.components.PDFDocument*, *page_number: int = 1*, *elements: Optional[ElementList] = None*, *show_info: bool = False*, *width: Optional[int] = None*, *height: Optional[int] = None*) → None

Visualises a PDFDocument, allowing you to inspect all the elements.

Will open a Matplotlib window showing the page_number. You can use the black buttons on the right of the toolbar to navigate through pages.

> **Warning:** In order to show you the actual PDF behind the elements, your document must be initialised with pdf_file_path, and your PDF must be at the given path. If this is not done, the background will be white.

Parameters

- **document** (PDFDocument) – The pdf document to visualise.
- **page_number** (*int*) – The page to visualise. Note you can change pages using the arrow keys in the visualisation window.
- **elements** (ElementList, *optional*) – Which elements of the document to visualise. Defaults to all of the elements in the document.
- **show_info** (*bool*) – Shows an additional window allowing you to click on PDFElements and see details about them. Default: False.
- **width** – (int, optional): The initial width of the visualisation window. Default: Screen width.
- **height** – (int, optional): The initial height of the visualisation window. Default: Screen height.

CHAPTER 4

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

4.1 [Unreleased]

4.1.1 Changed

- Removed unused PyYAML dependency (#262)

4.2 [0.10.2] - 2022-11-07

4.2.1 Changed

- Various dependency updates

4.3 [0.10.1] - 2021-10-12

4.3.1 Fixed

- The `visualise` function properly uses the *elements* parameter in order to filter visualised elements. (#256)

4.3.2 Changed

- Various dependency updates

4.4 [0.10.0] - 2021-07-01

- [BREAKING] Changes from using pyqt5 to using tkinter for the visualise tool. This means we don't need the python3-dev as a requirement, and seems to solve endless issues with pyqt5 not finding the correct qt bindings. This is a potential breaking change, although the visualise tool is only in the development version. No code changes are needed, but you will need tkinter installed for visualise to still work.
- Changed python version from 3.6 to 3.8 in `.readthedocs.yml`.

4.5 [0.9.0] - 2021-06-09

4.5.1 Changed

- Various dependency updates (matplotlib, pyqt5)
- Removed all but the tests dockerfile for simplicity. Use Docker BuildKit. We will no longer be pushing images to DockerHub on release. (#203)

4.6 [0.8.0] - 2021-05-12

4.6.1 Changed

- Various dependency updates
- Updated CI to avoid login issue (#182)

4.7 [0.7.0] - 2021-01-15

4.7.1 Changed

- Ensure we only accept LTTextBoxes at the top level (not LTTextLines) (#155)

4.8 [0.6.0] - 2020-12-11

4.8.1 Added

- Enabled dependabot which should help to keep packages up to date (#124)

4.8.2 Changed

- Various dependency updates

4.8.3 Fixed

- Fixed a typo in simple memo example in the documentation. (#121)

4.9 [0.5.0] - 2020-07-05

4.9.1 Added

- New functions on `ElementList`, `move_forwards_from` and `move_backwards_from`, to allow moving forwards and backwards from a certain element in the list easily. (#113)

4.9.2 Changed

- When the layout parameter all_texts is True, the text inside figures is now also returned as elements in the document. (#99)

4.9.3 Fixed

- Passing a tolerance less than the width/height of an element no longer causes an error. The tolerance is now capped at half the width/height of the element. (#103)

4.10 [0.4.0] - 2020-06-22

4.10.1 Added

- Added `__len__` and `__repr__` functions to the Section class. (#90)
- Added flag to `extract_simple_table` and `extract_table` functions to remove duplicate header rows. (#89)
- You can now specify `element_ordering` when instantiating a PDFDocument. This defaults to the old behaviour or left to right, top to bottom. (#95)

4.10.2 Changed

- Advanced layout analysis is now disabled by default. (#88)

4.11 [0.3.0] - 2020-05-14

4.11.1 Added

- Published to PyPI as py-pdf-parser.
- Documentation is now hosted here. (#71)
- Added new examples to the documentation. (#74)
- Font filtering now caches the elements by font. (#73) (updated in #78)
- Font filtering now caches the elements by font. (#73)
- The visualise tool now draws an outline around each section on the page. (#69) (updated in #80)

4.11.2 Changed

- This product is now complete enough for the needs of Optimor Ltd, however `jstockwin` is going to continue development as a personal project. The repository has been moved from `optimor/py-pdf-parser` to `jstockwin/py-pdf-parser`.

4.12 [0.2.0] - 2020-04-17

4.12.1 Added

- It is now possible to specify `font_size_precision` when instantiating a PDFDocument. This is the number of decimal places the font size will be rounded to. (#60)
- `extract_simple_table` now allows extracting tables with gaps, provided there is at least one full row and one full column. This is only the case if you pass `allow_gaps=True`, otherwise the original logic of raising an exception if there a gap remains. You can optionally pass a `reference_element` which must be in both a full row and a full column, this defaults to the first (top-left) element. (#57)

4.12.2 Changed

- Font sizes are now `float` not `int`. The `font_size_precision` in the additions defaults to 1, and as such all fonts will change to have a single decimal place. To keep the old behaviour, you can pass `font_size_precision=0` when instantiating your PDFDocument.

4.12.3 Fixed

- Improved performance of `extract_simple_table`, which is now much faster. (#65)

4.13 [0.1.0] - 2020-04-08

4.13.1 Added

- Initial version of the product. Note: The version is less than 1, so this product should not yet be considered stable. API changes and other breaking changes are possible, if not likely.

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W

X

Y