Camelot Documentation

Release 2.0.0

Vinayak Mehta

Jun 04, 2026

CONTENTS

Release v2.0.0. (*Installation*)

Camelot is a Python library that can help you extract tables from PDFs.

> **Note**
>
> You can also check out Excalibur, the web interface to Camelot.

Extract tables from PDFs in just a few lines of code:

Try it yourself in our interactive quickstart notebook.

Or check out a simple example using this pdf.

```
>>> import camelot
>>> tables = camelot.read_pdf('foo.pdf')
>>> tables
<TableList n=1>
>>> tables.export('foo.csv', f='csv', compress=True) # json, excel, html, markdown,␣
↪sqlite
>>> tables[0]
<Table shape=(7, 7)>
>>> tables[0].parsing_report
{
    'accuracy': 99.02,
    'whitespace': 12.24,
    'order': 1,
    'page': 1
}
>>> tables[0].to_csv('foo.csv') # to_json, to_excel, to_html, to_markdown, to_sqlite
>>> tables[0].df # get a pandas DataFrame!
```

Cycle Name	KI (1/km)	Distance (mi)	Percent Fuel Savings			
			Improved Speed	Decreased Accel	Eliminate Stops	Decreased Idle
2012_2	3.30	1.3	5.9%	9.5%	29.2%	17.4%
2145_1	0.68	11.2	2.4%	0.1%	9.5%	2.7%
4234_1	0.59	58.7	8.5%	1.3%	8.5%	3.3%
2032_2	0.17	57.8	21.7%	0.3%	2.7%	1.2%
4171_1	0.07	173.9	58.1%	1.6%	2.1%	0.5%

Camelot also comes packaged with a *command-line interface*!

Note

Camelot only works with text-based PDFs and not scanned documents. (As Tabula explains, "If you can click and drag to select text in your table in a PDF viewer, then your PDF is text-based".)

You can check out some frequently asked questions *here*.

CHAPTER

ONE

WHY CAMELOT?

- **Configurability**: Camelot gives you control over the table extraction process with *tweakable settings*.
- **Metrics**: You can discard bad tables based on metrics like accuracy and whitespace, without having to manually look at each table.
- **Output**: Each table is extracted into a **pandas DataFrame**, which seamlessly integrates into ETL and data analysis workflows. You can also export tables to multiple formats, which include CSV, JSON, Excel, HTML, Markdown, and Sqlite.

See *comparison with similar libraries and tools*.

CHAPTER

TWO

THE USER GUIDE

This part of the documentation begins with some background information about why Camelot was created, takes you through some implementation details, and then focuses on step-by-step instructions for getting the most out of Camelot.

2.1 Introduction

2.1.1 The Camelot Project

The PDF (Portable Document Format) was born out of The Camelot Project to create "a universal way to communicate documents across a wide variety of machine configurations, operating systems and communication networks". The goal was to make these documents viewable on any display and printable on any modern printers. The invention of the PostScript page description language, which enabled the creation of *fixed-layout* flat documents (with text, fonts, graphics, images encapsulated), solved this problem.

At a high level, PostScript defines instructions, such as "place this character at this *x,y* coordinate on a plane". Spaces can be *simulated* by placing characters relatively far apart. Extending from that, tables can be *simulated* by placing characters (which constitute words) in two-dimensional grids. A PDF viewer just takes these instructions and draws everything for the user to view. Since a PDF is just characters on a plane, there is no table data structure that can be extracted and used for analysis!

Sadly, a lot of today's open data is trapped in PDF tables.

2.1.2 Why another PDF table extraction library?

There are both open (Tabula, pdf-table-extract) and closed-source (smallpdf, PDFTables) tools that are widely used to extract tables from PDF files. They either give a nice output or fail miserably. There is no in between. This is not helpful since everything in the real world, including PDF table extraction, is fuzzy. This leads to the creation of ad-hoc table extraction scripts for each type of PDF table.

Camelot was created to offer users complete control over table extraction. If you can't get your desired output with the default settings, you can tweak them and get the job done!

Here is a comparison of Camelot's output with outputs from other open-source PDF parsing libraries and tools.

2.1.3 What's in a name?

As you can already guess, this library is named after The Camelot Project.

Fun fact: In the British comedy film Monty Python and the Holy Grail (and in the Arthurian legend depicted in the film), "Camelot" is the name of the castle where Arthur leads his men, the Knights of the Round Table, and then sets off elsewhere after deciding that it is "a silly place". Interestingly, the language in which this library is written (Python) was named after Monty Python.

2.1.4 Camelot License

2.2 Installation of dependencies

Note

as of `v1.0.0` ghostscript is replaced by pdfium as the default image conversion backend. This should make the library easier to install with just a pip install (on linux). The other image conversion backends can still be used and are now optional to install.

The optional dependency Ghostscript can be installed using your system's package manager or by running their installer.

2.2.1 OS-specific instructions

Ubuntu

```
$ apt install ghostscript
```

MacOS

```
$ brew install ghostscript
```

Note

You might encounter the problem that the ghostscript module cannot be found. This can be fixed with the following commands.

```
mkdir -p ~/lib
```

```
ln -s "$(brew --prefix gs)/lib/libgs.dylib" ~/lib
```

Windows

For Ghostscript, you can get the installer at their downloads page. And for Tkinter, you can download the ActiveTcl Community Edition from ActiveState.

2.2.2 Checks to see if dependencies are installed correctly

You can run the following checks to see if the dependencies were installed correctly.

For Ghostscript

Open the Python REPL and run the following:

For Ubuntu/MacOS

```
>>> from ctypes.util import find_library
>>> find_library("gs")
"libgs.so.9"
```

For Windows

```
>>> import ctypes
>>> from ctypes.util import find_library
>>> find_library("".join(("gsdll", str(ctypes.sizeof(ctypes.c_voidp) * 8), ".dll")))
<name-of-ghostscript-library-on-windows>
```

Check: The output of the `find_library` function should not be empty.

If the output is empty, then it's possible that the Ghostscript library is not available one of the `LD_LIBRARY_PATH/DYLD_LIBRARY_PATH/PATH` variables depending on your operating system. In this case, you may have to modify one of those path variables.

2.3 Installation

This part of the documentation covers the steps to install Camelot.

> **Note**
>
> as of `v1.0.0` ghostscript is replaced by pdfium as the default image conversion backend. This should make the library easier to install with just a pip install (on linux). The other image conversion backends can still be used and are now optional to install.

You can use one of the following methods to install Camelot:

2.3.1 pip

To install Camelot from PyPI using `pip`:

```
$ pip install "camelot-py"
```

> **Warning**

Camelot depends on `opencv-python-headless` (the GUI-less OpenCV build). If your environment already has the full `opencv-python` package installed, pip will let both sit side-by-side and the two shadow each other in `site-packages`, which leads to broken `import cv2` at runtime. Uninstall the conflicting package first:

```
$ pip uninstall opencv-python
$ pip install camelot-py
```

See issue #645.

2.3.2 conda

conda is a package manager and environment management system for the Anaconda distribution. It can be used to install Camelot from the `conda-forge` channel:

```
$ conda install -c conda-forge camelot-py
```

2.3.3 From the source code

After *installing the dependencies*, you can install Camelot from source by:

1. Cloning the GitHub repository.

```
$ git clone https://www.github.com/camelot-dev/camelot
```

2. And then simply using pip again.

```
$ cd camelot
$ pip install "."
```

2.3.4 Optional Dependencies

Additional dependencies for Camelot can be installed using the following options

- `[plot]` installs the python package `matplotlib` and is used for *visual debugging*.
- `[ghostscript]` installs the python package `ghostscript` and is used for the optional ghostscript backend.

Note that `[ghostscript]` only installs the python package `ghostscript`, which provides an interface to the Ghostscript C-API. Users must still download and install Ghostscript manually.

2.4 How It Works

This part of the documentation includes a high-level explanation of how Camelot extracts tables from PDF files.

You can choose between the following table parsing methods, *Stream*, *Lattice*, *Network* and *Hybrid*. Where *Hybrid* is a combination of the *Network* and *Lattice* parser.

> **Tip**
>
> For a side-by-side visual comparison of the parsers use our comparison notebook.

2.4.1 Stream

Stream can be used to parse tables that have whitespaces between cells to simulate a table structure. It is built on top of playa's PDFMiner-compatible functionality for grouping characters on a page into words and sentences, using margins.

1. Words on the PDF page are grouped into text rows based on their y axis overlaps.
2. Textedges are calculated and then used to guess interesting table areas on the PDF page. You can read Anssi Nurminen's master's thesis to know more about this table detection technique. [See pages 20, 35 and 40]
3. The number of columns inside each table area are then guessed. This is done by calculating the mode of number of words in each text row. Based on this mode, words in each text row are chosen to calculate a list of column x ranges.
4. Words that lie inside/outside the current column x ranges are then used to extend the current list of columns.
5. Finally, a table is formed using the text rows' y ranges and column x ranges and words found on the page are assigned to the table's cells based on their x and y coordinates.

2.4.2 Lattice

Lattice is more deterministic in nature, and it does not rely on guesses. It can be used to parse tables that have demarcated lines between cells, and it can automatically parse multiple tables present on a page.

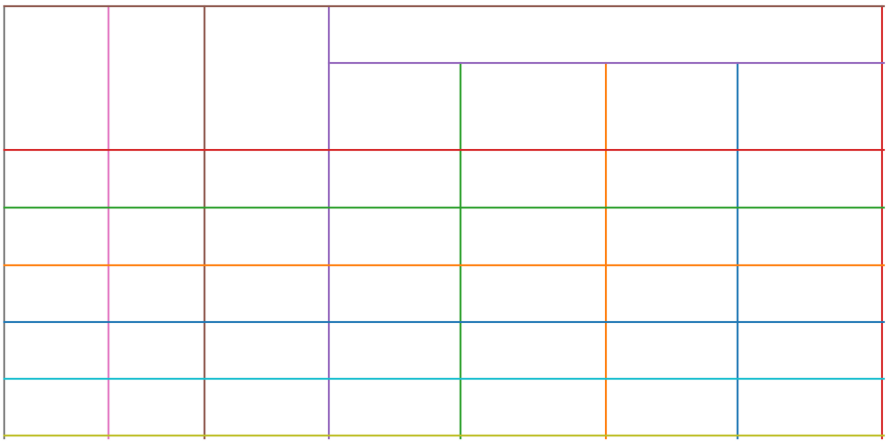
It starts by converting the PDF page to an image using an image conversion backend (default pdfium), and then processes it to get horizontal and vertical line segments by applying a set of morphological transformations (erosion and dilation) using OpenCV.

Note

That image-based (*raster*) line detection is the default. When a PDF draws its rules as native vector graphics, those exact line coordinates are already in the document — and they sometimes render too faintly for the rasteriser to pick up. The `engine='combined'` option (see *Line-detection engine: raster, combined, auto*) reads those vector lines directly and **unions** them into the rasterised line masks before the grid below is reconstructed, so faintly-ruled vector tables are still found. Because raster detection always runs first, the vector lines can only add to the result — `'combined'` is never worse than the default `'raster'`.

Let's see how Lattice processes the second page of this PDF, step-by-step.

1. Line segments are detected.



2. Line intersections are detected, by overlapping the detected line segments and "and"ing their pixel intensities.

Table 2-1. Simulated fuel savings from isolated cycle improvements

Cycle Name	KI (1/km)	Distance (mi)	Percent Fuel Savings			
			Improved Speed	Decreased Accel	Eliminate Stops	Decreased Idle
2012_2	3.30	1.3	5.9%	9.5%	29.2%	17.4%
2145_1	0.68	11.2	2.4%	0.1%	9.5%	2.7%
4234_1	0.59	58.7	8.5%	1.3%	8.5%	3.3%
2032_2	0.17	57.8	21.7%	0.3%	2.7%	1.2%
4171_1	0.07	173.9	58.1%	1.6%	2.1%	0.5%

2-1 extends the analysis from eliminating stops for the five example cycles and exam

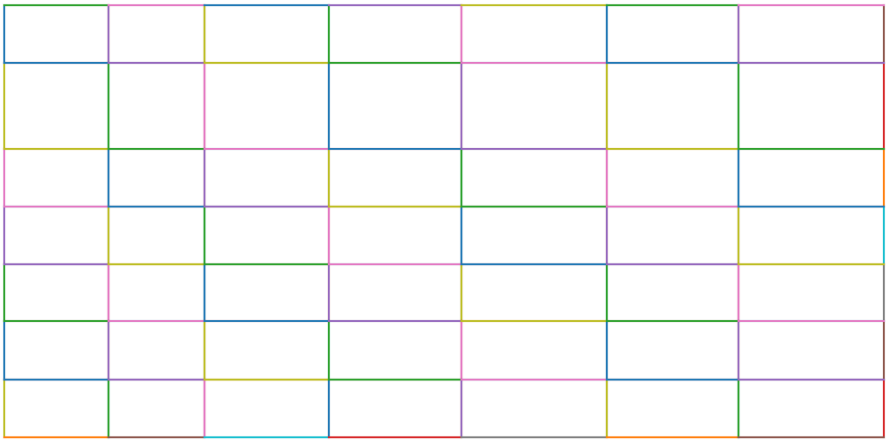
3. Table boundaries are computed by overlapping the detected line segments again, this time by "or"ing their pixel intensities.

Table 2-1. Simulated fuel savings from isolated cycle improvements

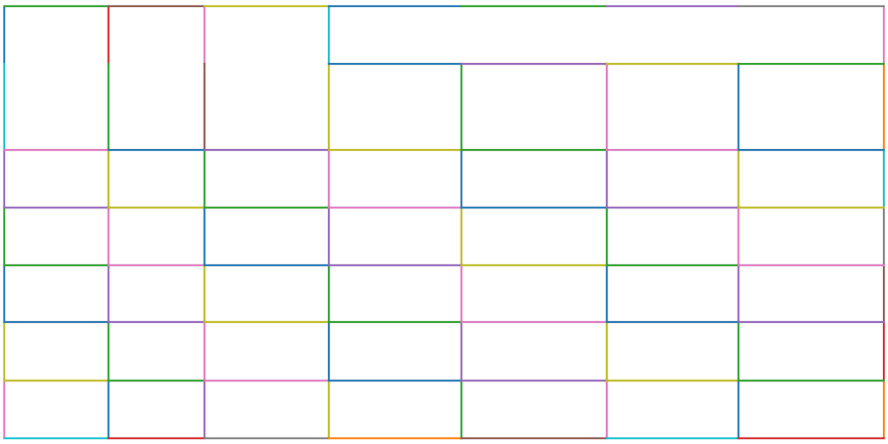
Cycle Name	KI (1/km)	Distance (mi)	Percent Fuel Savings			
			Improved Speed	Decreased Accel	Eliminate Stops	Decreased Idle
2012_2	3.30	1.3	5.9%	9.5%	29.2%	17.4%
2145_1	0.68	11.2	2.4%	0.1%	9.5%	2.7%
4234_1	0.59	58.7	8.5%	1.3%	8.5%	3.3%
2032_2	0.17	57.8	21.7%	0.3%	2.7%	1.2%
4171_1	0.07	173.9	58.1%	1.6%	2.1%	0.5%

2-1 extends the analysis from eliminating stops for the five example cycles and exam

4. Since dimensions of the PDF page and its image vary, the detected table boundaries, line intersections, and line segments are scaled and translated to the PDF page's coordinate space, and a representation of the table is created.



5. Spanning cells are detected using the line segments and line intersections.



6. Finally, the words found on the page are assigned to the table's cells based on their *x* and *y* coordinates.

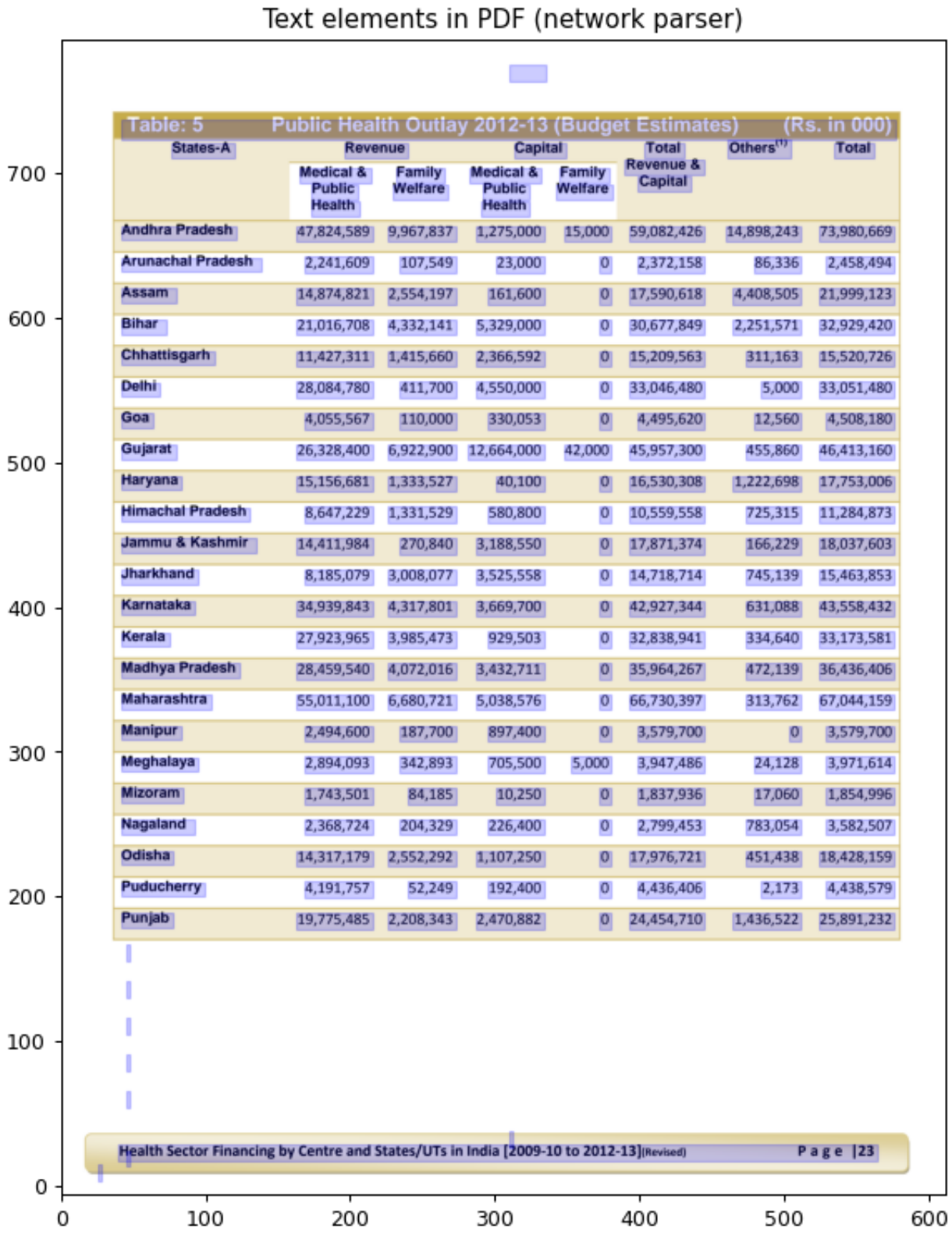
2.4.3 Network

> **Tip**
>
> The mechanism of the Network and Hybrid parser can best be understood by using the following notebook:

The network parser is text-based: it relies on the bounding boxes of the text elements encoded in the .pdf document to identify patterns indicative of a table.

The parser starts by enumerating the bounding boxes of every text element on the page — typically horizontal-running boxes (the bulk of the text) and occasional vertical-running boxes (rare in most documents).

The plot below shows the bounding boxes of all the text elements found on the page (here the `network` parser running on a public-health table):



1. The network parser starts by identifying common horizontal or vertical coordinate alignments across these text elements. In other words it looks for bounding box rectangles which either share the same top, center, or bottom coordinates (horizontal axis), or the same left, right, or middle coordinates (vertical axis). See the generate

method.

Once the parser found these alignments, it performs some pruning to only keep text elements that are part of a network - they have connections along both axis The idea is that it's not enough for two elements to be aligned to belong to a table, for instance the lines of text in this paragraph are all left-aligned, but they do not form a network. The pruning is done iteratively, see "remove_unconnected_edges" method.

Once the network is pruned, the parser counts how many alignments each text element belongs to. The text element with the most connections is the starting point — the *seed* — of the next step. Finally, the parser measures how far the alignments are from one another, to determine a plausible search zone around each cell for the next stage of growing the table. See "compute_plausible_gaps" method.

2. n the next step, the parser iteratively "grows" a table, starting from the seed identified in the previous step. The bounding box is initialized with the bounding box of the seed, then it iteratively searches for text elements that are close to the bounding box, then grows the table to ingest them, until there are no more text elements to ingest. The two steps are:

Search: create a search bounding box by expanding the current table bounding box in all directions, based on the plausible gap numbers determined above. Grow: if a networked text element is found in this search area, expand the table bounding box so that it includes this new element.

The search area and the table bounding box grow starting from the seed. See method "search_table_body".

3. Headers are often aligned differently from the rest of the table. To account for this, the network parser searches for text elements that are good candidates for a header section: these text elements are just above the bounding box of the body of the table, and they fit within the rows identified in the table body. See the method "search_header_from_body_bbox".
4. Words that lie inside/outside the current column *x* ranges are then used to extend the current list of columns.
5. There are sometimes multiple tables on one page. So once a first table is identified, all the text edges it contains are removed, and the algorithm is repeated until no new network is identified.

2.4.4 Hybrid

The hybrid parser aims to combine the strengths of the Network parser (identifying cells based on text alignments) and of the Lattice parser (relying on solid lines to determine tables rows and columns boundaries).

1. Hybrid calls both parsers, to get a) the standard table parse, b) the coordinates of the rows and columns boundaries, and c) the table boundaries (or contour).
2. If there are areas in the document where both lattice and network found a table, the hybrid parser uses the results from network, but enhances them based on the rows/columns boundaries identified by lattice in the area. Because lattice uses the solid lines detected on the document, the coordinates for b) and c) detected by Lattice are generally more precise. See the "_merge_bbox_analysis" method.

2.4.5 ML (Table Transformer)

`flavor='ml'` is an **optional** neural backend for the tables the heuristic parsers find hardest — dense **borderless** tables — and, with OCR, for **scanned / image-only** PDFs. It is opt-in because it pulls PyTorch: `pip install 'camelot-py[ml]'` (add `[ocr]` for scans).

Its guiding rule is **the model supplies the structure, the page supplies the text**:

1. The page is rendered to an image. A Table Transformer (TATR) detection model finds the table region(s); a second TATR model recognises the **structure** — rows, columns and spanning cells — as bounding boxes.
2. Those boxes are turned into the same column/row/spanning grid the other parsers build, then mapped from image space back to PDF coordinates.

3. Each cell's **text** is filled from the PDF's own text layer (the exact characters), reusing the same assignment step as every other flavor — so the model never emits a character of cell text and **cannot hallucinate or alter a value**. For scanned pages with no text layer, `ocr='auto'` reads the text from the rendered image instead (still geometry + recognised text, never invented cells).

Because the structure half runs on the page image, `flavor='ml'` is the only parser that works without a text layer at all.

Borderless accuracy

On the borderless FinTabNet.c benchmark (545 financial PDFs, `bench/benchmark_fintabnet.py`), the model backend clears the ceiling the heuristic parsers plateau at:

flavor	F1	TEDS	row	col
ml	0.750	0.371	0.235	0.570
network	0.725	0.200	0.109	0.220
hybrid (combined)	0.658	0.198	0.109	0.217

(`TEDS`/`row`/`col` from `bench/_metrics.py`; `TEDS` here is a difflib cell-text proxy, not exact tree-edit distance, so the absolute values are conservative — the point is the **relative** gap: `ml` roughly doubles borderless `TEDS` over `network`/`hybrid`.) The trade-off is speed (~1 s/page with a GPU vs tens of ms for `network`) and the optional PyTorch dependency, which is why it is opt-in rather than the default.

2.5 Quickstart

In a hurry to extract tables from PDFs? This document gives a good introduction to help you get started with Camelot. You can also check out our quickstart notebook.

2.5.1 Read the PDF

Reading a PDF to extract tables with Camelot is very simple.

Begin by importing the Camelot module:

```
>>> import camelot
```

Now, let's try to read a PDF. (You can check out the PDF used in this example here.) Since the PDF has a table with clearly demarcated lines, we will use the *Lattice* method here.

> **Note**
>
> *Lattice* is used by default. You can use *Stream* with `flavor='stream'`, *Network* with `flavor='network'`, *Hybrid* with `flavor='hybrid'`, or `flavor='auto'` to let Camelot probe the first page and choose between `lattice` and `network`. When `auto` is selected a `UserWarning` names the chosen flavor.

```
>>> tables = camelot.read_pdf('foo.pdf')
>>> tables
<TableList n=1>
```

You can also pass raw PDF bytes or any binary stream (`io.BytesIO`, an open `'rb'` file, `requests` response `.raw`) wherever a filepath is accepted — useful for PDFs that arrive over HTTP without hitting disk first. See the *Reading PDFs from memory* section in the advanced guide for the full pattern.

Now, we have a *TableList* object called `tables`, which is a list of *Table* objects. We can get everything we need from this object.

We can access each table using its index. From the code snippet above, we can see that the `tables` object has only one table, since `n=1`. Let's access the table using the index `0` and take a look at its `shape`.

```
>>> tables[0]
<Table shape=(7, 7)>
```

Let's print the parsing report.

```
>>> print(tables[0].parsing_report)
{
    'accuracy': 99.02,
    'whitespace': 12.24,
    'confidence': 0.87,
    'order': 1,
    'page': 1
}
```

Woah! The accuracy is top-notch and there is less whitespace, which means the table was most likely extracted correctly. The `confidence` value is a single `[0, 1]` score computed as `(accuracy / 100) * (1 - whitespace / 100)` — convenient for "keep tables above X" filtering in production pipelines without having to combine the two raw fields yourself. You can access the table as a pandas DataFrame by using the *table* object's `df` property.

```
>>> tables[0].df
```

Cycle Name	KI (1/km)	Distance (mi)	Percent Fuel Savings			
			Improved Speed	Decreased Accel	Eliminate Stops	Decreased Idle
2012_2	3.30	1.3	5.9%	9.5%	29.2%	17.4%
2145_1	0.68	11.2	2.4%	0.1%	9.5%	2.7%
4234_1	0.59	58.7	8.5%	1.3%	8.5%	3.3%
2032_2	0.17	57.8	21.7%	0.3%	2.7%	1.2%
4171_1	0.07	173.9	58.1%	1.6%	2.1%	0.5%

Looks good! You can now export the table as a CSV file using its *to_csv()* method. Alternatively you can use *to_json()*, *to_excel()* *to_html()* *to_markdown()* or *to_sqlite()* methods to export the table as JSON, Excel, HTML files or a sqlite database respectively.

```
>>> tables[0].to_csv('foo.csv')
```

This will export the table as a CSV file at the path specified. In this case, it is `foo.csv` in the current directory.

You can also export all tables at once, using the *tables* object's *export()* method.

```
>>> tables.export('foo.csv', f='csv')
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot --format csv --output foo.csv lattice foo.pdf
```

This will export all tables as CSV files at the path specified. Alternatively, you can use `f='json'`, `f='excel'`, `f='html'`, `f='markdown'` or `f='sqlite'`.

Note

The *export()* method exports files with a `page-*-table-*` suffix. In the example above, the single table in the list will be exported to `foo-page-1-table-1.csv`. If the list contains multiple tables, multiple CSV files will be created. To avoid filling up your path with multiple files, you can use `compress=True`, which will create a single ZIP file at your path with all the CSV files.

Note

Camelot handles rotated PDF pages automatically. As an exercise, try to extract the table out of this PDF.

2.5.2 Specify page numbers

By default, Camelot only uses the first page of the PDF to extract tables. To specify multiple pages, you can use the `pages` keyword argument:

```
>>> camelot.read_pdf('your.pdf', pages='1,2,3')
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot --pages 1,2,3 lattice your.pdf
```

The `pages` keyword argument accepts pages as comma-separated string of page numbers. You can also specify page ranges — for example, `pages=1,4-10,20-30` or `pages=1,4-10,20-end`.

When `parallel=True`, Camelot processes pages concurrently using one worker per CPU. Bound the worker count with `cpu_count=N` (defaults to all cores; clamped to `[1, multiprocessing.cpu_count()]`):

```
>>> camelot.read_pdf('long.pdf', pages='all', parallel=True, cpu_count=4)
```

If different pages need different settings (per-page `table_areas`, a different `flavor` on one page, etc.), use the `per_page` keyword argument. See the *Per-page parameter overrides* section in the advanced guide.

2.5.3 Reading encrypted PDFs

To extract tables from encrypted PDF files you must provide a password when calling *read_pdf()*.

```
>>> tables = camelot.read_pdf('foo.pdf', password='userpass')
>>> tables
<TableList n=1>
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot --password userpass lattice foo.pdf
```

Camelot supports PDFs with all encryption types supported by playa. This might require installing PyCryptodome. An exception is thrown if the PDF cannot be read. This may be due to no password being provided, an incorrect password, or an unsupported encryption algorithm.

Further encryption support may be added in future, however in the meantime if your PDF files are using unsupported encryption algorithms you are advised to remove encryption before calling `read_pdf()`. This can been successfully achieved with third-party tools such as QPDF.

```
$ qpdf --password=<PASSWORD> --decrypt input.pdf output.pdf
```

Ready for more? Check out the *advanced* section.

2.6 Advanced Usage

This page covers some of the more advanced configurations for *Lattice* and *Stream*.

2.6.1 Process background lines

To detect line segments, *Lattice* needs the lines that make the table to be in the foreground. Here's an example of a table with lines in the background:

Table 6.1: The highlights of RRE-II coverage (till 11 March, 2010)

State	Date	Halt stations	Halt days	Persons directly reached (in lakh)	Persons trained	Persons counseled	Persons tested for HIV
Delhi	1.12.2009	8	17	1.29	3,665	2,409	1,000
Rajasthan	2.12.2009 to 19.12.2009						
Gujarat	20.12.2009 to 3.1.2010	6	13	6.03	3,810	2,317	1,453
Maharashtra	4.01.2010 to 1.2.2010	13	26	1.27	5,680	9,027	4,153
Karnataka	2.2.2010 to 22.2.2010	11	19	1.80	5,741	3,658	3,183
Kerala	23.2.2010 to 11.3.2010	9	17	1.42	3,559	2,173	855
Total		**47**	**92**	**11.81**	**22,455**	**19,584**	**10,644**

It includes visitors to train exhibition and those reached through outreach activities

Source: PDF

To process background lines, you can pass `process_background=True`.

```
>>> tables = camelot.read_pdf('background_lines.pdf', process_background=True)
>>> tables[1].df
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot lattice -back background_lines.pdf
```

State	Date	Halt stations	Halt days	Persons directly reached(in lakh)	Persons trained	Persons counseled	Persons testedfor HIV
Delhi	1.12.2009	8	17	1.29	3,665	2,409	1,000
Rajasthan	2.12.2009 to 19.12.2009						
Gujarat	20.12.2009 to 3.1.2010	6	13	6.03	3,810	2,317	1,453
Maharashtra	4.01.2010 to 1.2.2010	13	26	1.27	5,680	9,027	4,153
Karnataka	2.2.2010 to 22.2.2010	11	19	1.80	5,741	3,658	3,183
Kerala	23.2.2010 to 11.3.2010	9	17	1.42	3,559	2,173	855
Total		47	92	11.81	22,455	19,584	10,644

2.6.2 Bridge gaps in ruled lines

When a Lattice-flavoured PDF's table is drawn with ruled lines that don't quite meet at corners (a common artefact of older scanned-then-redrawn forms), the detected grid drops the affected rows or columns. The `iterations` keyword argument dilates the line mask to close those gaps — but dilation alone also *thickens* every line, which in turn pushes the outer ruled lines outward and adds spurious extra rows above and below the real table.

Pair it with the new `erode_iterations` keyword to perform a **morphological closing** (dilate then erode of equal count): gaps are bridged without changing the line mask's overall size, so the detected grid stays right.

```
>>> # Bridges line gaps without thickening
>>> tables = camelot.read_pdf(
...     'broken_lines.pdf',
...     flavor='lattice',
...     iterations=1,
...     erode_iterations=1,
... )
```

`erode_iterations` defaults to `0` (fully backward-compatible with the long-standing dilate-only behaviour). Bump it together with `iterations` only when you've confirmed the legacy behaviour leaves phantom rows around your table.

2.6.3 Line-detection engine: raster, combined, auto

By default *Lattice* finds a table's ruled lines by **rasterising** the page (rendering it to an image) and detecting lines with OpenCV. That works well for scanned or image-based tables, but a PDF that draws its rules as *native vector graphics* carries the exact line coordinates already — and those rules sometimes render faintly or anti-aliased, so the rasteriser misses them.

The `engine` keyword lets you choose how lines are detected:

- `'raster'` *(default)* — OpenCV on the rendered page. The long-standing behaviour.

- `'combined'` — run raster detection **and** union in the ruled lines read straight from the PDF's vector graphics before the grid is reconstructed. A table whose rules are vector strokes is then found even when it renders faintly.
- `'auto'` — use `'combined'` when the page actually carries vector ruled lines, otherwise fall back to `'raster'`.
- `'vector'` — detect tables purely from the PDF's vector ruled lines, skipping rasterisation entirely. The fastest engine (no page render, no OpenCV), for PDFs whose tables are drawn with real vector strokes. A page with no vector ruled lines yields no tables, so prefer `'auto'` / `'combined'` for mixed documents.

```
>>> # Recover a faintly-ruled vector table that 'raster' misses
>>> tables = camelot.read_pdf(
...     'vector_ruled.pdf',
...     flavor='lattice',
...     engine='combined',
... )
```

`'combined'` is **safe to try on any lattice PDF**: raster detection always runs first, so the vector lines can only *add* to what was found, never remove it. On a PDF whose rules the rasteriser already detects cleanly, `engine='combined'` returns exactly the same tables as `engine='raster'`.

The same keyword works with `flavor='hybrid'`, where it drives the lattice half of the hybrid parser:

```
>>> tables = camelot.read_pdf(
...     'mixed_layout.pdf',
...     flavor='hybrid',
...     engine='combined',
... )
```

2.6.4 Filter out noise tables

Detection sometimes returns small or low-quality "tables" — a stray single cell, a mostly-empty region, a heading mistaken for a 1x1 grid. *TableList.filter()* keeps only the tables that pass the thresholds you give and returns a new *TableList*; extraction itself is unchanged.

```
>>> tables = camelot.read_pdf('noisy.pdf')
>>> # keep tables with at least 2 rows and 2 columns
>>> real = tables.filter(min_rows=2, min_columns=2)
>>> # ...or filter on parsing quality, and compose freely
>>> good = tables.filter(min_accuracy=90).filter(max_whitespace=50)
```

Every threshold defaults to a no-op (`min_rows=1`, `min_columns=1`, `min_accuracy=0`, `max_whitespace=100`), so a legitimate single-row or single-column table is never dropped unless you ask for it. `accuracy` and `whitespace` are the same 0–100 values reported in *Table.parsing_report*.

2.6.5 Visual debugging

> **Note**
>
> Visual debugging using `plot()` requires matplotlib which is an optional dependency. You can install it using `$ pip install camelot-py[plot]`.

You can use the *plot()* method to generate a matplotlib plot of various elements that were detected on the PDF page while processing it. This can help you select table areas, column separators and debug bad table outputs, by tweaking different configuration parameters.

You can specify the type of element you want to plot using the `kind` keyword argument. The generated plot can be saved to a file by passing a `filename` keyword argument. The following plot types are supported:

- 'text'
- 'grid'
- 'contour'
- 'line'
- 'joint'
- 'textedge'

> **Note**
>
> 'line' and 'joint' can only be used with *Lattice* and 'textedge' can only be used with *Stream*.

Let's generate a plot for each type using this PDF as an example. First, let's get all the tables out.

```
>>> tables = camelot.read_pdf('foo.pdf')
>>> tables
<TableList n=1>
```

text

Let's plot all the text present on the table's PDF page.

```
>>> camelot.plot(tables[0], kind='text').show()
```

> **Tip**
>
> Here's how you can do the same with the *command-line interface*.
>
> ```
> $ camelot lattice -plot text foo.pdf
> ```

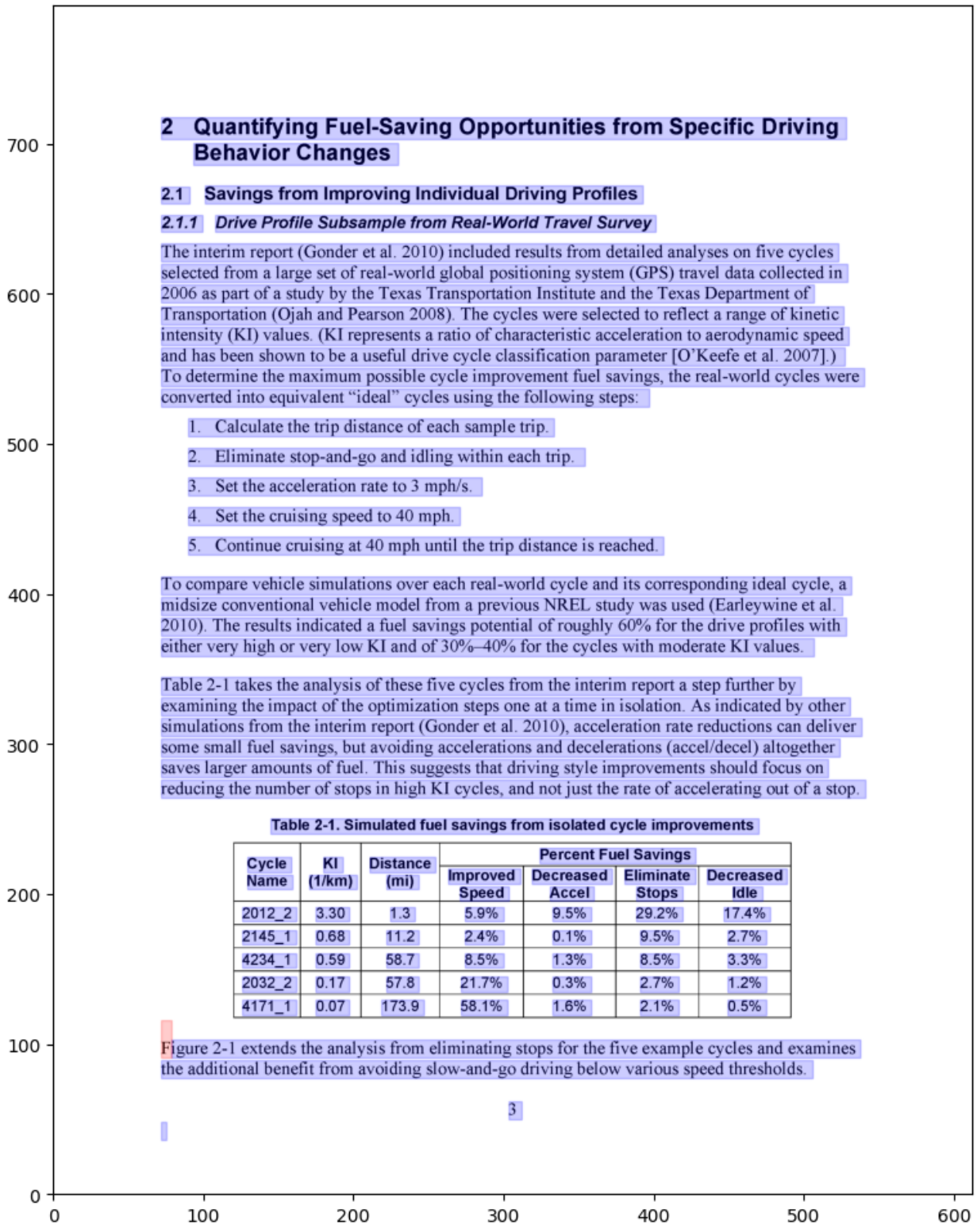
This, as we shall later see, is very helpful with *Stream* for noting table areas and column separators, in case Stream does not guess them correctly.

> **Note**
>
> The *x-y* coordinates shown above change as you move your mouse cursor on the image, which can help you note coordinates.

table

Let's plot the table (to see if it was detected correctly or not). This plot type, along with contour, line and joint is useful for debugging and improving the extraction output, in case the table wasn't detected correctly. (More on that later.)

```
>>> camelot.plot(tables[0], kind='grid').show()
```



2 Quantifying Fuel-Saving Opportunities from Specific Driving Behavior Changes

2.1 Savings from Improving Individual Driving Profiles

2.1.1 Drive Profile Subsample from Real-World Travel Survey

The interim report (Gonder et al. 2010) included results from detailed analyses on five cycles selected from a large set of real-world global positioning system (GPS) travel data collected in 2006 as part of a study by the Texas Transportation Institute and the Texas Department of Transportation (Ojah and Pearson 2008). The cycles were selected to reflect a range of kinetic intensity (KI) values. (KI represents a ratio of characteristic acceleration to aerodynamic speed and has been shown to be a useful drive cycle classification parameter [O'Keefe et al. 2007].) To determine the maximum possible cycle improvement fuel savings, the real-world cycles were converted into equivalent "ideal" cycles using the following steps:

1. Calculate the trip distance of each sample trip.
2. Eliminate stop-and-go and idling within each trip.
3. Set the acceleration rate to 3 mph/s.
4. Set the cruising speed to 40 mph.
5. Continue cruising at 40 mph until the trip distance is reached.

To compare vehicle simulations over each real-world cycle and its corresponding ideal cycle, a midsize conventional vehicle model from a previous NREL study was used (Earleywine et al. 2010). The results indicated a fuel savings potential of roughly 60% for the drive profiles with either very high or very low KI and of 30%–40% for the cycles with moderate KI values.

Table 2-1 takes the analysis of these five cycles from the interim report a step further by examining the impact of the optimization steps one at a time in isolation. As indicated by other simulations from the interim report (Gonder et al. 2010), acceleration rate reductions can deliver some small fuel savings, but avoiding accelerations and decelerations (accel/decel) altogether saves larger amounts of fuel. This suggests that driving style improvements should focus on reducing the number of stops in high KI cycles, and not just the rate of accelerating out of a stop.

Table 2-1. Simulated fuel savings from isolated cycle improvements

Cycle Name	KI (1/km)	Distance (mi)	Percent Fuel Savings			
			Improved Speed	Decreased Accel	Eliminate Stops	Decreased Idle
2012_2	3.30	1.3	5.9%	9.5%	29.2%	17.4%
2145_1	0.68	11.2	2.4%	0.1%	9.5%	2.7%
4234_1	0.59	58.7	8.5%	1.3%	8.5%	3.3%
2032_2	0.17	57.8	21.7%	0.3%	2.7%	1.2%
4171_1	0.07	173.9	58.1%	1.6%	2.1%	0.5%

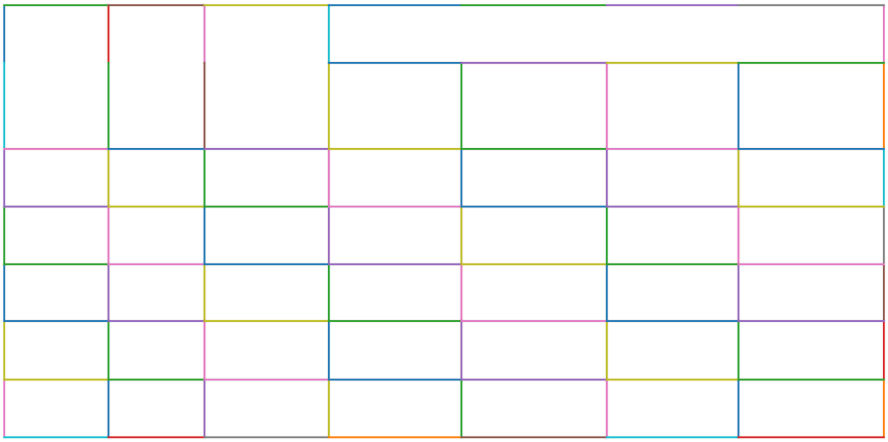
Figure 2-1 extends the analysis from eliminating stops for the five example cycles and examines the additional benefit from avoiding slow-and-go driving below various speed thresholds.

3

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot lattice -plot grid foo.pdf
```



The table is perfect!

contour

Now, let's plot all table boundaries present on the table's PDF page.

```
>>> camelot.plot(tables[0], kind='contour').show()
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot lattice -plot contour foo.pdf
```

line

Cool, let's plot all line segments present on the table's PDF page.

```
>>> camelot.plot(tables[0], kind='line').show()
```

Tip

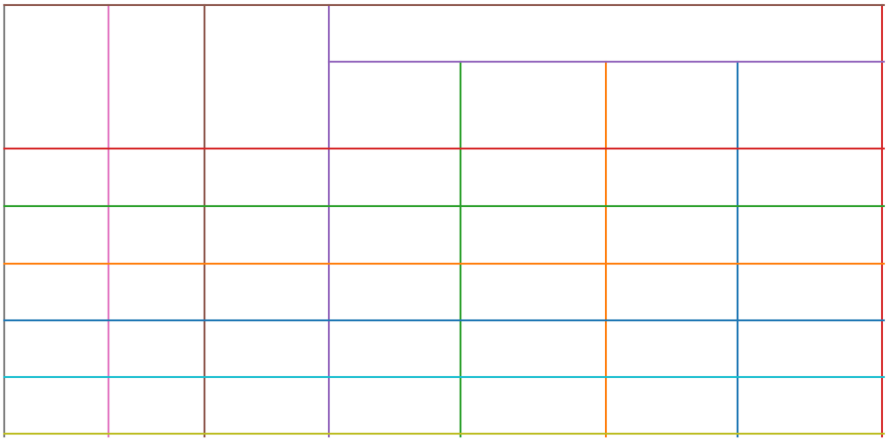
Here's how you can do the same with the *command-line interface*.

```
$ camelot lattice -plot line foo.pdf
```

Table 2-1. Simulated fuel savings from isolated cycle improvements

Cycle Name	KI (1/km)	Distance (mi)	Percent Fuel Savings			
			Improved Speed	Decreased Accel	Eliminate Stops	Decreased Idle
2012_2	3.30	1.3	5.9%	9.5%	29.2%	17.4%
2145_1	0.68	11.2	2.4%	0.1%	9.5%	2.7%
4234_1	0.59	58.7	8.5%	1.3%	8.5%	3.3%
2032_2	0.17	57.8	21.7%	0.3%	2.7%	1.2%
4171_1	0.07	173.9	58.1%	1.6%	2.1%	0.5%

2-1 extends the analysis from eliminating stops for the five example cycles and exam



joint

Finally, let's plot all line intersections present on the table's PDF page.

```
>>> camelot.plot(tables[0], kind='joint').show()
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot lattice -plot joint foo.pdf
```

Table 2-1. Simulated fuel savings from isolated cycle improvements

Cycle Name	KI (1/km)	Distance (mi)	Percent Fuel Savings			
			Improved Speed	Decreased Accel	Eliminate Stops	Decreased Idle
2012_2	3.30	1.3	5.9%	9.5%	29.2%	17.4%
2145_1	0.68	11.2	2.4%	0.1%	9.5%	2.7%
4234_1	0.59	58.7	8.5%	1.3%	8.5%	3.3%
2032_2	0.17	57.8	21.7%	0.3%	2.7%	1.2%
4171_1	0.07	173.9	58.1%	1.6%	2.1%	0.5%

2-1 extends the analysis from eliminating stops for the five example cycles and exam

textedge

You can also visualize the textedges found on a page by specifying `kind='textedge'`. To know more about what a "textedge" is, you can see pages 20, 35 and 40 of Anssi Nurminen's master's thesis:

```
>>> camelot.plot(tables[0], kind='textedge').show()
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot stream -plot textedge foo.pdf
```

2.6.6 Specify table areas

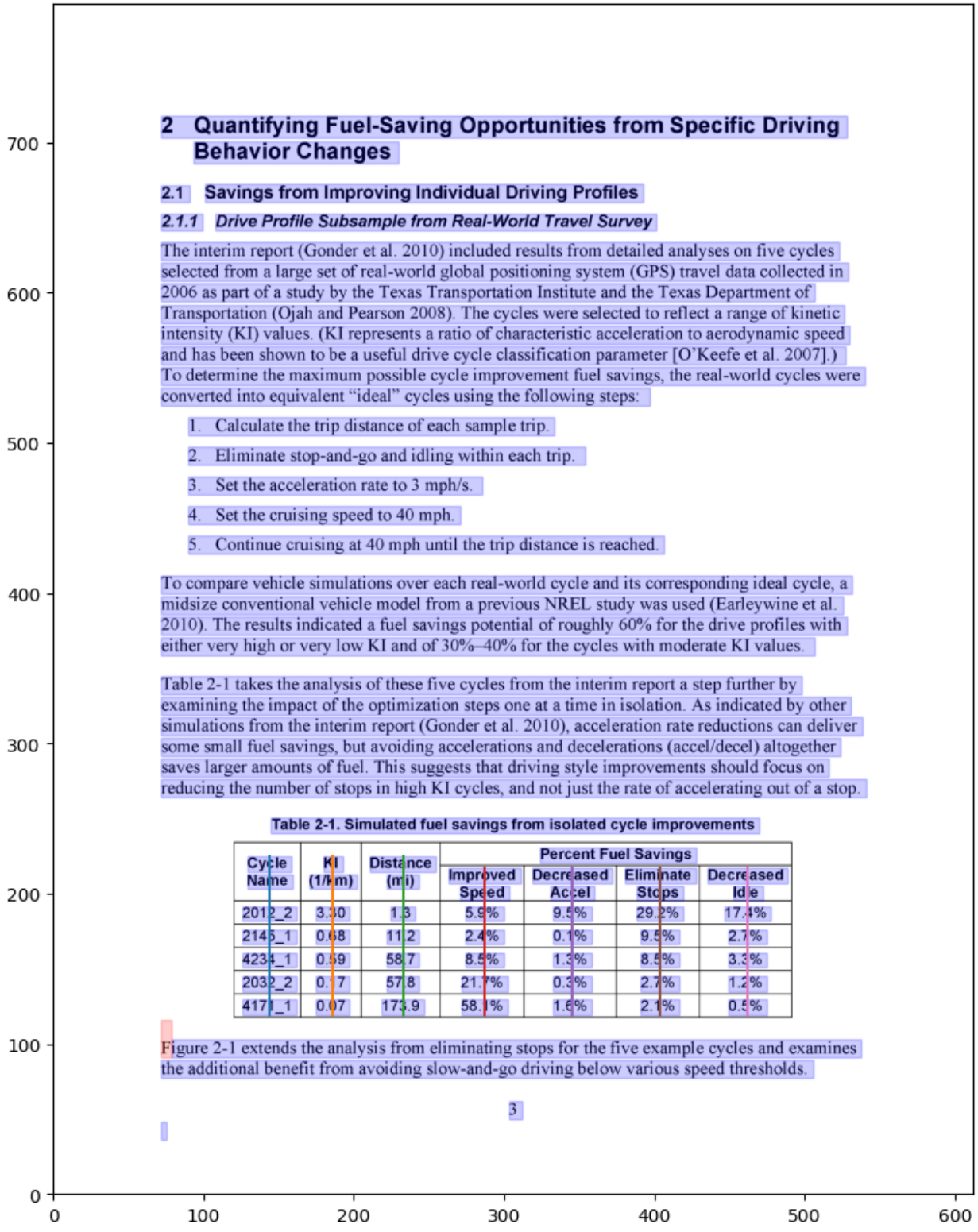
In cases such as these, it can be useful to specify exact table boundaries. You can plot the text on this page and note the top left and bottom right coordinates of the table.

Table areas that you want camelot to analyze can be passed as a list of comma-separated strings to *read_pdf()*, using the `table_areas` keyword argument.

```
>>> tables = camelot.read_pdf('table_areas.pdf', flavor='stream', table_areas=['316,499,
↪566,337'])
>>> tables[0].df
```

Tip

Here's how you can do the same with the *command-line interface*.


2 Quantifying Fuel-Saving Opportunities from Specific Driving Behavior Changes
2.1 Savings from Improving Individual Driving Profiles
2.1.1 Drive Profile Subsample from Real-World Travel Survey
The interim report (Gonder et al. 2010) included results from detailed analyses on five cycles selected from a large set of real-world global positioning system (GPS) travel data collected in 2006 as part of a study by the Texas Transportation Institute and the Texas Department of Transportation (Ojah and Pearson 2008). The cycles were selected to reflect a range of kinetic intensity (KI) values. (KI represents a ratio of characteristic acceleration to aerodynamic speed and has been shown to be a useful drive cycle classification parameter [O'Keefe et al. 2007].) To determine the maximum possible cycle improvement fuel savings, the real-world cycles were converted into equivalent "ideal" cycles using the following steps:
1. Calculate the trip distance of each sample trip.
2. Eliminate stop-and-go and idling within each trip.
3. Set the acceleration rate to 3 mph/s.
4. Set the cruising speed to 40 mph.
5. Continue cruising at 40 mph until the trip distance is reached.
To compare vehicle simulations over each real-world cycle and its corresponding ideal cycle, a midsize conventional vehicle model from a previous NREL study was used (Earleywine et al. 2010). The results indicated a fuel savings potential of roughly 60% for the drive profiles with either very high or very low KI and of 30%–40% for the cycles with moderate KI values.
Table 2-1 takes the analysis of these five cycles from the interim report a step further by examining the impact of the optimization steps one at a time in isolation. As indicated by other simulations from the interim report (Gonder et al. 2010), acceleration rate reductions can deliver some small fuel savings, but avoiding accelerations and decelerations (accel/decel) altogether saves larger amounts of fuel. This suggests that driving style improvements should focus on reducing the number of stops in high KI cycles, and not just the rate of accelerating out of a stop.
Table 2-1. Simulated fuel savings from isolated cycle improvements
Cycle Name
KI (1/km)
Distance (mi)
Percent Fuel Savings
Improved Speed
Decreased Accel
Eliminate Stops
Decreased Idle
2012_2 3.30 1.3 5.9% 9.5% 29.2% 17.4%
2145_1 0.68 11.2 2.4% 0.1% 9.5% 2.7%
4234_1 0.59 58.7 8.5% 1.3% 8.5% 3.3%
2032_2 0.17 57.8 21.7% 0.3% 2.7% 1.2%
4171_1 0.07 173.9 58.1% 1.6% 2.1% 0.5%
Figure 2-1 extends the analysis from eliminating stops for the five example cycles and examines the additional benefit from avoiding slow-and-go driving below various speed thresholds.
3
0
100
200
300
400
500
600
0
100
200
300
400
500
600

```
$ camelot stream -T 316,499,566,337 table_areas.pdf
```

	One Withholding
Payroll Period	Allowance
Weekly	$71.15
Biweekly	142.31
Semimonthly	154.17
Monthly	308.33
Quarterly	925.00
Semiannually	1,850.00
Annually	3,700.00
Daily or Miscellaneous	14.23
(each day of the payroll period)	

Note

`table_areas` accepts strings of the form x1,y1,x2,y2 where (x1, y1) -> top-left and (x2, y2) -> bottom-right in PDF coordinate space. In PDF coordinate space, the bottom-left corner of the page is the origin, with coordinates (0, 0).

2.6.7 Specify table regions

However there may be cases like [1] and [2], where the table might not lie at the exact coordinates every time but in an approximate region.

You can use the `table_regions` keyword argument to *`read_pdf()`* to solve for such cases. When `table_regions` is specified, camelot will only analyze the specified regions to look for tables.

```
>>> tables = camelot.read_pdf('table_regions.pdf', table_regions=['170,370,560,270'])
>>> tables[0].df
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot lattice -R 170,370,560,270 table_regions.pdf
```

Età dell'Assicuratoall'epoca del decesso	Misura % dimaggiorazione
18-75	1,00%
76-80	0,50%
81 in poi	0,10%

2.6.8 Specify column separators

In cases like these, where the text is very close to each other, it is possible that camelot may guess the column separators' coordinates incorrectly. To correct this, you can explicitly specify the *x* coordinate for each column separator by plotting the text on the page.

You can pass the column separators as a list of comma-separated strings to *read_pdf()*, using the `columns` keyword argument.

In case you passed a single column separators string list, and no table area is specified, the separators will be applied to the whole page. When a list of table areas is specified and you need to specify column separators as well, **the length of both lists should be equal**. Each table area will be mapped to each column separators' string using their indices.

For example, if you have specified two table areas, `table_areas=['12,54,43,23', '20,67,55,33']`, and only want to specify column separators for the first table, you can pass an empty string for the second table in the column separators' list like this, `columns=['10,120,200,400', '']`.

Let's get back to the *x* coordinates we got from plotting the text that exists on this PDF, and get the table out!

```
>>> tables = camelot.read_pdf('column_separators.pdf', flavor='stream', columns=['72,95,
↪209,327,442,529,566,606,683'])
>>> tables[0].df
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot stream -C 72,95,209,327,442,529,566,606,683 column_separators.pdf
```

…	…	…	…	…	…	…	…	…	…
LICENSE				PREMISE					
NUMBER TYPE DBA NAME			LICENSEE NAME	AD-DRESS	CITY	ST	ZIP	PHONE NUMBER	EX-PIRES
…	…	…	…	…	…	…	…	…	…

Ah! Since playa (the PDFMiner-compatible layout engine Camelot now uses) merged the strings, "NUMBER", "TYPE" and "DBA NAME", all of them were assigned to the same cell. Let's see how we can fix this in the next section.

2.6.9 Split text along separators

To deal with cases like the output from the previous section, you can pass `split_text=True` to *read_pdf()*, which will split any strings that lie in different cells but have been assigned to a single cell (as a result of being merged together by playa, the PDFMiner-compatible layout engine).

```
>>> tables = camelot.read_pdf('column_separators.pdf', flavor='stream', columns=['72,95,
↪209,327,442,529,566,606,683'], split_text=True)
>>> tables[0].df
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot -split stream -C 72,95,209,327,442,529,566,606,683 column_separators.pdf
```

…	…	…	…	…	…	…	…	…	…
LI-CENSE				PREMISE					
NUM-BER	TYPE	DBA NAME	LICENSEE NAME	AD-DRESS	CITY	ST	ZIP	PHONE NUM-BER	EX-PIRES
…	…	…	…	…	…	…	…	…	…

2.6.10 Flag superscripts and subscripts

There might be cases where you want to differentiate between the text and superscripts or subscripts, like this PDF.

Jammu and Kashmir	11.72	4.49	-	-	7.23	0.66
Jharkhand	-	-	-	-	-	-
Karnataka	22.44	19.59	-	-	2.86	1.22
Kerala	29.03	24.91[2]	-	-	4.11	1.77
Madhya Pradesh	27.13	23.57	-	-	3.56	0.38
Maharashtra	30.47	26.07	-	-	4.39	0.21
Manipur	2.17	1.61	-	0.26	0.29	0.08

In this case, the text that other tools return, will be `24.912`. This is relatively harmless when that decimal point is involved. But when it isn't there, you'll be left wondering why the results of your data analysis are 10x bigger!

You can solve this by passing `flag_size=True`, which will enclose the superscripts and subscripts with `<s></s>`, based on font size, as shown below.

```
>>> tables = camelot.read_pdf('superscript.pdf', flavor='stream', flag_size=True)
>>> tables[0].df
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot -flag stream superscript.pdf
```

…	…	…	…	…	…	…	…	…	…	…
Karnataka	22.44	19.59	•	•	2.86	1.22	•	0.89	•	0.69
Kerala	29.03	24.91<s>2</s>	•	•	4.11	1.77	•	0.48	•	1.45
Madhya Pradesh	27.13	23.57	•	•	3.56	0.38	•	1.86	•	1.28
…	…	…	…	…	…	…	…	…	…	…

2.6.11 Strip characters from text

You can strip unwanted characters like spaces, dots and newlines from a string using the `strip_text` keyword argument. Take a look at this PDF as an example, the text at the start of each row contains a lot of unwanted spaces, dots and newlines.

```
>>> tables = camelot.read_pdf('12s0324.pdf', flavor='stream', strip_text=' .\n')
>>> tables[0].df
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot -strip ' .\n' stream 12s0324.pdf
```

…	…	…	…	…	…	…	…	…	…
Forcible rape	17.5	2.6	14.9	17.2	2.5	14.7	–	–	–
Robbery	102.1	25.5	76.6	90.0	22.9	67.1	12.1	2.5	9.5
Aggravated assault	338.4	40.1	298.3	264.0	30.2	233.8	74.4	9.9	64.5
Property crime	1,396 .4	338 .7	1,057 .7	875 .9	210 .8	665 .1	608 .2	127 .9	392 .6
Burglary	240.9	60.3	180.6	205.0	53.4	151.7	35.9	6.9	29.0
…	…	…	…	…	…	…	…	…	…

The `strip_text` argument also accepts a **list or tuple of substrings**, in which case each whole substring is removed wherever it appears (rather than each individual character). This is the right mode when you want to strip multi-character markers like footnote references without nicking lone brackets / digits elsewhere in the cell:

```
>>> # Per-character (long-standing behaviour): strips any of '[', ']', '1', '2'
>>> camelot.read_pdf('doc.pdf', strip_text='[12]')

>>> # Per-substring (new in 2.0): strips only the literal markers '[1]' and '[2]',
>>> # leaves stray '[' or ']' alone.
>>> camelot.read_pdf('doc.pdf', strip_text=['[1]', '[2]'])
```

2.6.12 Replace text in cells

Where `strip_text` can only **remove** characters or substrings, `replace_text` lets you **rewrite** them. It accepts a `dict` mapping substrings to their replacements, applied to every cell's text just before assignment.

A common motivating example: words that PDF text extraction has split across a soft line break end up concatenated without a space. Use `replace_text` to turn *" n"* (space + newline) into a single space:

```
>>> tables = camelot.read_pdf('doc.pdf', replace_text={' \n': ' '})
```

You can normalise unit names, expand abbreviations, or fix systematic OCR-style mistakes in the same call:

```
>>> tables = camelot.read_pdf(
...     'doc.pdf',
...     replace_text={'kw': 'kW', 'kva': 'kVA', 'µ': 'micro'},
... )
```

Keys are matched as literal substrings (regex metacharacters are escaped, so `"."` matches a literal dot). When several keys could match at the same position, the longest one wins, so `{"abc": "X", "ab": "Y"}` replaces `"abc"` with `"X"` rather than producing `"Yc"`. Empty keys are ignored.

`replace_text` works with every flavor (`lattice`, `stream`, `network`, `hybrid`) and stacks cleanly with `strip_text` — stripping runs first, then replacement.

2.6.13 Per-page parameter overrides

When a single PDF has pages with different table layouts — say a cover page with no table, two body pages with stream-flavour text columns, and an appendix with a ruled lattice table — calling *read_pdf()* once per page-group works but means re-opening the PDF and re-running parser setup each time.

The `per_page` keyword argument lets you keep the global kwargs and override just the ones that need to change for specific pages:

```
>>> tables = camelot.read_pdf(
...     'report.pdf',
...     pages='1-3',
...     flavor='stream',
...     split_text=True,
...     per_page={2: {'table_areas': ['120, 210, 400, 90']}},
... )
```

Here pages 1 and 3 use `flavor='stream'` with `split_text=True`. Page 2 uses both **and** the page-specific `table_areas`.

The `per_page` keys are 1-indexed page numbers (int or str). The values are dicts of any kwarg otherwise valid for *read_pdf()*, including a per-page `flavor`. Unknown kwargs and unknown flavors raise the same errors as if they were passed globally, named by their offending page.

2.6.14 Reading PDFs from memory

Beyond filesystem paths and URLs, *read_pdf()* accepts in-memory PDF content directly: `bytes`, `bytearray`, an `io.BytesIO`, or any binary stream with a `.read()` method (an open `"rb"` file, a `requests` response's `.raw`, etc.):

```
>>> import io, requests, camelot
>>>
>>> # Bytes you already have in memory
>>> data = open('doc.pdf', 'rb').read()
```

(continues on next page)

(continued from previous page)

```
>>> camelot.read_pdf(data)
>>>
>>> # An io.BytesIO
>>> camelot.read_pdf(io.BytesIO(data))
>>>
>>> # Straight from an HTTP response
>>> resp = requests.get('https://example.org/doc.pdf')
>>> camelot.read_pdf(io.BytesIO(resp.content))
```

Camelot writes the bytes to a temporary file once internally (so the Lattice OpenCV image-conversion backend keeps working unchanged) and removes the temp file when the handler is closed. For file-like inputs the read position is preserved so the caller can keep using the same stream afterwards.

2.6.15 Improve guessed table areas

While using *Stream*, automatic table detection can fail for PDFs like this one. That's because the text is relatively far apart vertically, which can lead to shorter textedges being calculated.

Note

To know more about how textedges are calculated to guess table areas, you can see pages 20, 35 and 40 of Anssi Nurminen's master's thesis.

Let's see the table area that is detected by default.

```
>>> tables = camelot.read_pdf('edge_tol.pdf', flavor='stream')
>>> camelot.plot(tables[0], kind='contour').show()
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot stream -plot contour edge_tol.pdf
```

To improve the detected area, you can increase the `edge_tol` (default: 50) value to counter the effect of text being placed relatively far apart vertically. Larger `edge_tol` will lead to longer textedges being detected, leading to an improved guess of the table area. Let's use a value of 500.

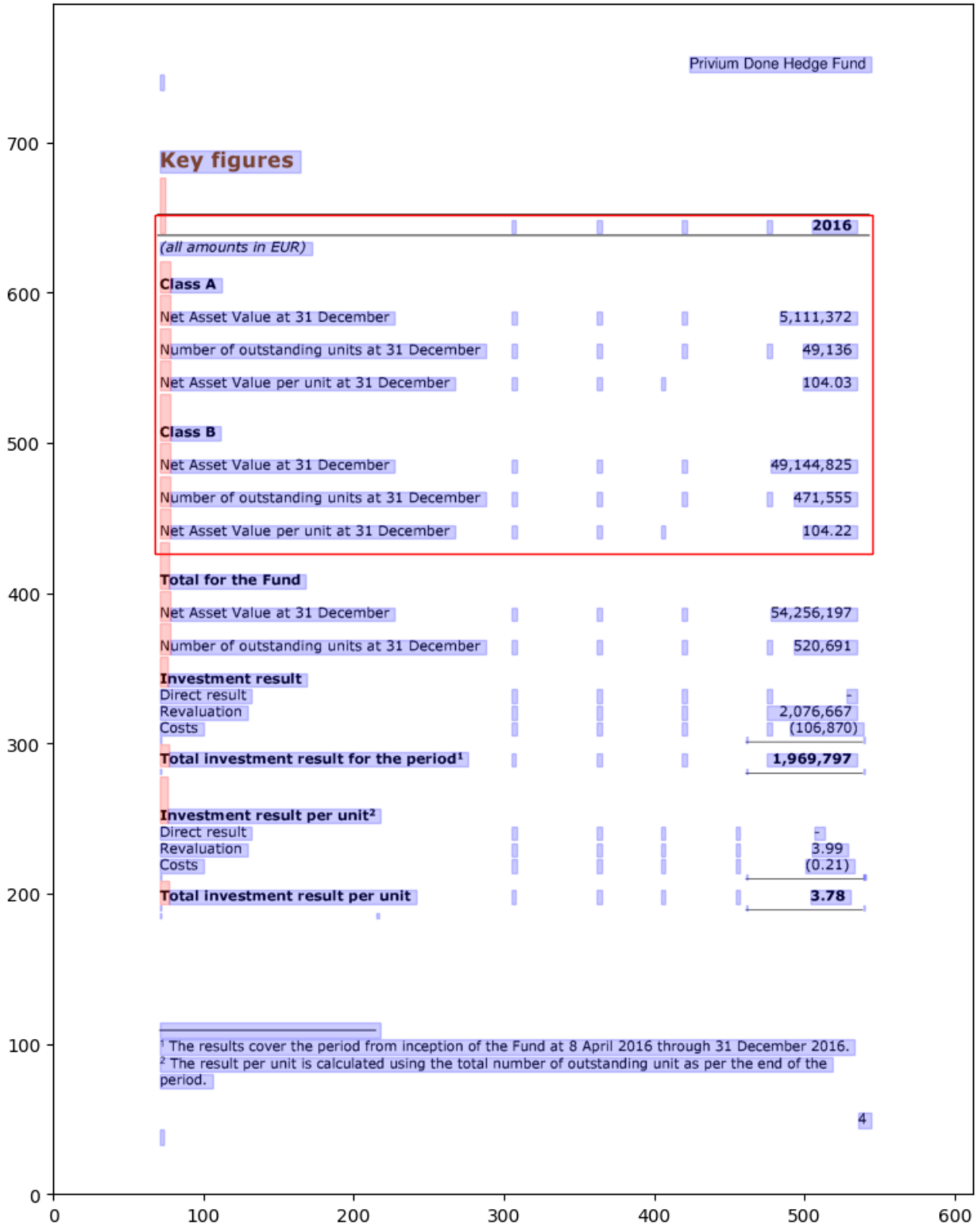
```
>>> tables = camelot.read_pdf('edge_tol.pdf', flavor='stream', edge_tol=500)
>>> camelot.plot(tables[0], kind='contour').show()
```

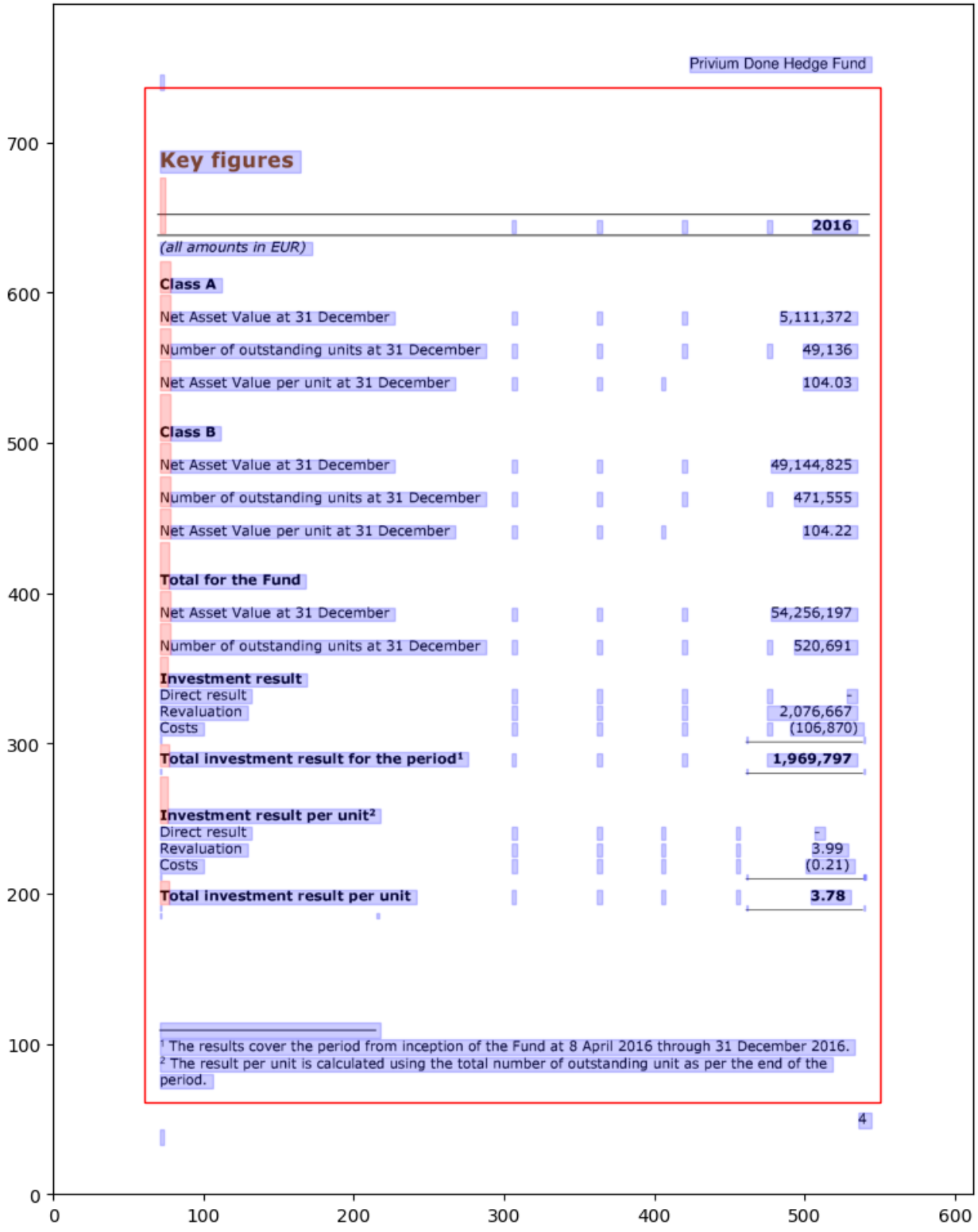
Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot stream -e 500 -plot contour edge_tol.pdf
```

As you can see, the guessed table area has improved!


Privium Done Hedge Fund
Key figures
2016
(all amounts in EUR)
Class A
Net Asset Value at 31 December 5,111,372
Number of outstanding units at 31 December 49,136
Net Asset Value per unit at 31 December 104.03
Class B
Net Asset Value at 31 December 49,144,825
Number of outstanding units at 31 December 471,555
Net Asset Value per unit at 31 December 104.22
Total for the Fund
Net Asset Value at 31 December 54,256,197
Number of outstanding units at 31 December 520,691
Investment result
Direct result -
Revaluation 2,076,667
Costs (106,870)
Total investment result for the period¹ 1,969,797
Investment result per unit²
Direct result -
Revaluation 3.99
Costs (0.21)
Total investment result per unit 3.78
¹ The results cover the period from inception of the Fund at 8 April 2016 through 31 December 2016.
² The result per unit is calculated using the total number of outstanding unit as per the end of the period.
4
0
100
200
300
400
500
600
700


Privium Done Hedge Fund
Key figures
2016
(all amounts in EUR)
Class A
Net Asset Value at 31 December 5,111,372
Number of outstanding units at 31 December 49,136
Net Asset Value per unit at 31 December 104.03
Class B
Net Asset Value at 31 December 49,144,825
Number of outstanding units at 31 December 471,555
Net Asset Value per unit at 31 December 104.22
Total for the Fund
Net Asset Value at 31 December 54,256,197
Number of outstanding units at 31 December 520,691
Investment result
Direct result -
Revaluation 2,076,667
Costs (106,870)
Total investment result for the period[1] 1,969,797
Investment result per unit[2]
Direct result -
Revaluation 3.99
Costs (0.21)
Total investment result per unit 3.78
[1] The results cover the period from inception of the Fund at 8 April 2016 through 31 December 2016.
[2] The result per unit is calculated using the total number of outstanding unit as per the end of the period.
4
0
100
200
300
400
500
600
700

2.6.16 Improve guessed table rows

You can pass `row_tol=<+int>` to group the rows closer together, as shown below.

```
>>> tables = camelot.read_pdf('group_rows.pdf', flavor='stream')
>>> tables[0].df
```

Clave		Clave			Clave	
	Nombre Entidad			Nombre Municipio		Nombre Localidad
Entidad		Municipio			Localidad	
01	Aguascalientes	001	Aguascalientes		0094	Granja Adelita
01	Aguascalientes	001	Aguascalientes		0096	Agua Azul
01	Aguascalientes	001	Aguascalientes		0100	Rancho Alegre

```
>>> tables = camelot.read_pdf('group_rows.pdf', flavor='stream', row_tol=10)
>>> tables[0].df
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot stream -r 10 group_rows.pdf
```

Clave	Nombre Entidad	Clave		Nombre Municipio	Clave	Nombre Localidad
Entidad		Municipio			Localidad	
01	Aguascalientes	001	Aguascalientes		0094	Granja Adelita
01	Aguascalientes	001	Aguascalientes		0096	Agua Azul
01	Aguascalientes	001	Aguascalientes		0100	Rancho Alegre

2.6.17 Detect short lines

There might be cases while using *Lattice* when smaller lines don't get detected. The size of the smallest line that gets detected is calculated by dividing the PDF page's dimensions with a scaling factor called `line_scale`. By default, its value is 40.

As you can guess, the larger the `line_scale`, the smaller the size of lines getting detected.

Warning

Making `line_scale` very large (>150) will lead to text getting detected as lines.

Here's a PDF where small lines separating the the headers don't get detected with the value of 15.

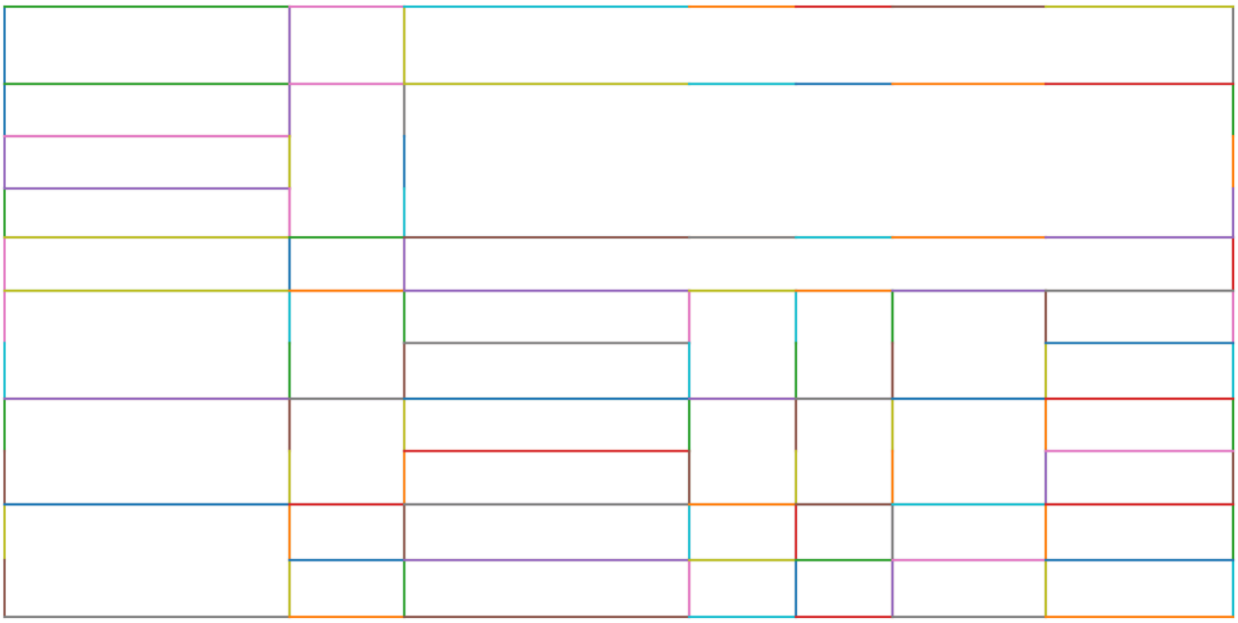
Investigations	No. of HHs	Age/Sex/ Physiological Group	Preva-lence	C.I*	Relative Precision	Sample size per State
Anthropometry	2400	All the available individuals				
Clinical Examination						
History of morbidity						
Diet survey	1200	All the individuals partaking meals in the HH				
Blood Pressure #	2400	Men (≥ 18yrs)	10%	95%	20%	1728
		Women (≥ 18 yrs)				1728
Fasting blood glucose	2400	Men (≥ 18 yrs)	5%	95%	20%	1825
		Women (≥ 18 yrs)				1825
Knowledge & Practices on HTN & DM	2400	Men (≥ 18 yrs)	-	-	-	1728
	2400	Women (≥ 18 yrs)	-	-	-	1728

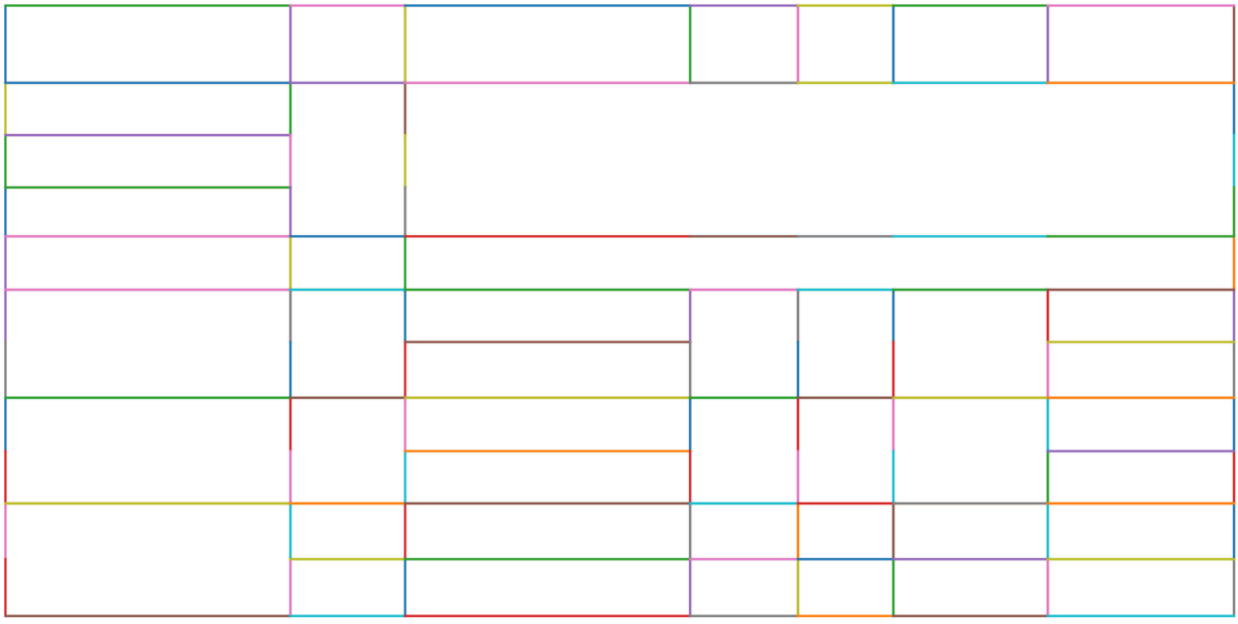
Let's plot the table for this PDF.

```
>>> tables = camelot.read_pdf('short_lines.pdf')
>>> camelot.plot(tables[0], kind='grid').show()
```

Clearly, the smaller lines separating the headers, couldn't be detected. Let's try with `line_scale=40`, and plot the table again.

```
>>> tables = camelot.read_pdf('short_lines.pdf', line_scale=40)
>>> camelot.plot(tables[0], kind='grid').show()
```





Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot lattice -scale 40 -plot grid short_lines.pdf
```

Voila! camelot can now see those lines. Let's get our table.

```
>>> tables[0].df
```

Investigations	No. ofHHs	Age/Sex/Physiological Group	Preva-lence	C.I*	RelativePre-cision	Sample sizeper State
Anthropometry	2400	All . . .				
Clinical Examination						
History of morbidity						
Diet survey	1200	All . . .				
Blood Pressure #	2400	Men (18yrs)	10%	95%	20%	1728
		Women (18 yrs)				1728
Fasting blood glucose	2400	Men (18 yrs)	5%	95%	20%	1825
		Women (18 yrs)				1825
Knowledge &Practices on HTN &DM	2400	Men (18 yrs)	•	•	•	1728
	2400	Women (18 yrs)	•	•	•	1728

2.6.18 Shift text in spanning cells

By default, the *Lattice* method shifts text in spanning cells, first to the left and then to the top, as you can observe in the output table above. However, this behavior can be changed using the `shift_text` keyword argument. Think of it as setting the *gravity* for a table — it decides the direction in which the text will move and finally come to rest.

`shift_text` expects a list with one or more characters from the following set: `('', 'l', 'r', 't', 'b')`, which are then applied *in order*. The default, as we discussed above, is `['l', 't']`.

We'll use the PDF from the previous example. Let's pass `shift_text=['']`, which basically means that the text will experience weightlessness! (It will remain in place.)

```
>>> tables = camelot.read_pdf('short_lines.pdf', line_scale=40, shift_text=[''])
>>> tables[0].df
```

Investigations	No. of HHs	Age/Sex/ Physiological Group	Preva- lence	C.I*	Relative Precision	Sample size per State
Anthropometry	2400	All the available individuals				
Clinical Examination						
History of morbidity						
Diet survey	1200	All the individuals partaking meals in the HH				
Blood Pressure #	2400	Men (≥ 18yrs)	10%	95%	20%	1728
		Women (≥ 18 yrs)				1728
Fasting blood glucose	2400	Men (≥ 18 yrs)	5%	95%	20%	1825
		Women (≥ 18 yrs)				1825
Knowledge & Practices on HTN & DM	2400	Men (≥ 18 yrs)	-	-	-	1728
	2400	Women (≥ 18 yrs)	-	-	-	1728

Investigations	No. ofHHs	Age/Sex/Physiological Group	Preva- lence	C.I*	RelativePre- cision	Sample sizeper State
Anthropometry						
Clinical Examination	2400		All …			
History of morbidity						
Diet survey	1200		All …			
		Men (18yrs)				1728
Blood Pressure #	2400	Women (18 yrs)	10%	95%	20%	1728
		Men (18 yrs)				1825
Fasting blood glucose	2400	Women (18 yrs)	5%	95%	20%	1825
Knowledge &Practices on HTN &	2400	Men (18 yrs)	•	•	•	1728
DM	2400	Women (18 yrs)	•	•	•	1728

No surprises there — it did remain in place (observe the strings “2400” and “All the available individuals”). Let’s pass `shift_text=['r', 'b']` to set the *gravity* to right-bottom and move the text in that direction.

```
>>> tables = camelot.read_pdf('short_lines.pdf', line_scale=40, shift_text=['r', 'b'])
>>> tables[0].df
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot lattice -scale 40 -shift r -shift b short_lines.pdf
```

Investigations	No. ofHHs	Age/Sex/Physiological Group	Preva-lence	C.I*	RelativePre-cision	Sample sizeper State
Anthropometry						
Clinical Examination						
History of morbidity	2400					All …
Diet survey	1200					All …
		Men (18yrs)				1728
Blood Pressure #	2400	Women (18 yrs)	10%	95%	20%	1728
		Men (18 yrs)				1825
Fasting blood glucose	2400	Women (18 yrs)	5%	95%	20%	1825
	2400	Men (18 yrs)	•	•	•	1728
Knowledge &Practices on HTN &DM	2400	Women (18 yrs)	•	•	•	1728

2.6.19 Copy text in spanning cells

You can copy text in spanning cells when using *Lattice*, in either the horizontal or vertical direction, or both. This behavior is disabled by default.

`copy_text` expects a list with one or more characters from the following set: `('v', 'h')`, which are then applied *in order*.

Let's try it out on this PDF. First, let's check out the output table to see if we need to use any other configuration parameters.

```
>>> tables = camelot.read_pdf('copy_text.pdf')
>>> tables[0].df
```

Sl. No.	Name of State/UT	Name of District	Disease/ Illness	No. of Cases	No. of Deaths	Date of start of outbreak	Date of report-ing	Current Status	...
1	Kerala	Kollam	i. Food Poi-soning	19	0	31/12/13	03/01/14	Under control	...
2	Maha-rashtra	Beed	i. Dengue & Chikun-gunya i	11	0	03/01/14	04/01/14	Under control	...
3	Odisha	Kala-handi	iii. Food Poi-soning	42	0	02/01/14	03/01/14	Under control	...
4	West Bengal	West Me-dinipur	iv. Acute Di-arrhoeal Disease	145	0	04/01/14	05/01/14	Under control	...
		Birb-hum	v. Food Poi-soning	199	0	31/12/13	31/12/13	Under control	...
		Howrah	vi. Viral Hepatitis A &E	85	0	26/12/13	27/12/13	Under surveil-lance	...

We don't need anything else. Now, let's pass `copy_text=['v']` to copy text in the vertical direction. This can save you some time by not having to add this step in your cleaning script!

```
>>> tables = camelot.read_pdf('copy_text.pdf', copy_text=['v'])
>>> tables[0].df
```

Tip

Here's how you can do the same with the *command-line interface*.

```
$ camelot lattice -copy v copy_text.pdf
```

Sl. No.	Name of State/UT	Name of District	Disease/ Illness	No. of Cases	No. of Deaths	Date of start of outbreak	Date of reporting	Current Status	...
1	Kerala	Kollam	i. Food Poisoning	19	0	31/12/13	03/01/14	Under control	...
2	Maharashtra	Beed	i. Dengue & Chikungunya	11	0	03/01/14	04/01/14	Under control	...
3	Odisha	Kalahandi	iii. Food Poisoning	42	0	02/01/14	03/01/14	Under control	...
4	West Bengal	West Medinipur	iv. Acute Diarrhoeal Disease	145	0	04/01/14	05/01/14	Under control	...
4	West Bengal	Birbhum	v. Food Poisoning	199	0	31/12/13	31/12/13	Under control	...
4	West Bengal	Howrah	vi. Viral Hepatitis A &E	85	0	26/12/13	27/12/13	Under surveillance	...

2.6.20 Tweak layout generation

Camelot is built on top of playa's PDFMiner-compatible functionality for grouping characters on a page into words and sentences. In some cases (such as #170 and #215), the layout engine can group characters that should belong to the same sentence into separate sentences.

To deal with such cases, you can tweak the layout engine's LAParams kwargs to improve layout generation, by passing the keyword arguments as a dict using `layout_kwargs` in *`read_pdf()`*. `playa.miner` mirrors PDFMiner.six's `LAParams`, so the upstream PDFMiner.six docs still describe what each parameter does.

```
>>> tables = camelot.read_pdf('foo.pdf', layout_kwargs={'detect_vertical': False})
```

2.6.21 Use alternative image conversion backends

When using the *Lattice* flavor, camelot uses `pdfium` to convert PDF pages to images for line recognition. You can still use `ghostscript` after installing it. You can specify which image conversion backend you want to use with

```
>>> tables = camelot.read_pdf(filename, backend="pdfium")  # default
>>> tables = camelot.read_pdf(filename, backend="ghostscript")
```

Note

`ghostscript` is replaced by `pdfium` as the default image conversion backend in `v1.0.0`.

If you face issues with `pdfium`, `ghostscript` and `poppler`, you can supply your own image conversion backend.

```
>>> class ConversionBackend(object):
>>>     def convert(pdf_path, png_path):
```

(continues on next page)

(continued from previous page)

```
>>>         # read pdf page from pdf_path
>>>         # convert pdf page to image
>>>         # write image to png_path
>>>         pass
>>>
>>> tables = camelot.read_pdf(filename, backend=ConversionBackend())
```

2.6.22 Working with image-based / scanned PDFs

Camelot extracts tables by reading the PDF's text operators — fonts, positions, kerning. For PDFs that are **image-only** (scanned pages saved to PDF, faxed forms, photos exported to PDF) there is no text to read; every "table" is just pixels and Camelot will report zero tables found.

The recommended workflow is to **add a text layer first, then run Camelot**. OCRmyPDF is a mature, dedicated tool for exactly this — it wraps Tesseract OCR and overlays the recognised text on the original page images, so the output PDF reads exactly like the input but now has selectable, searchable, extractable text underneath.

Install once:

```
$ pipx install ocrmypdf  # or: pip install ocrmypdf
```

…then run it in front of Camelot:

```
$ ocrmypdf scan.pdf scan-ocr.pdf
$ camelot lattice --output tables.csv scan-ocr.pdf
```

…or as a Python pipeline:

```
>>> import subprocess
>>> import camelot
>>> subprocess.run(["ocrmypdf", "scan.pdf", "scan-ocr.pdf"], check=True)
>>> tables = camelot.read_pdf("scan-ocr.pdf", flavor="lattice")
```

A few practical notes:

- **Mixed PDFs** (some pages text, some scanned) are handled by default — `ocrmypdf` will skip pages that already have a text layer unless you pass `--force-ocr`.
- **Languages other than English** need the corresponding Tesseract language pack installed (e.g. `apt install tesseract-ocr-deu` for German), then `ocrmypdf -l deu scan.pdf scan-ocr.pdf`.
- **Table-friendly OCR** benefits from a higher resolution and from preserving the original image. `ocrmypdf --image-dpi 300 --redo-ocr` is a reasonable default for documents whose scans are fuzzy.
- **Quality**: lattice-style ruled tables typically survive OCR well because the ruling lines are pixel-perfect; stream-style borderless tables depend heavily on how well Tesseract aligns the per-cell text — try `flavor="auto"` or `flavor="hybrid"` and inspect `Table.parsing_report` (especially the `confidence` field) to pick the better path.

Why isn't OCR built into Camelot? Tesseract is a heavyweight system dependency (binary install + language packs, hundreds of MB), and OCR quality is non-deterministic across versions — bundling it would make the install story much worse for the majority of users who already have text PDFs. Keeping OCR as a separate preprocessing step lets `ocrmypdf` handle the OCR concerns (image preprocessing, language detection, page rotation, etc.) and Camelot focus on the post-OCR text-to-table conversion.

For a full discussion see issue #14 and PR #209.

2.7 How Camelot compares to other tools

This page compares Camelot to the most common open-source PDF table-extraction libraries you're likely to evaluate alongside it. The goal isn't to claim Camelot is best for every PDF — different tools win on different inputs — it's to help you pick the right tool for your corpus quickly.

If you've already used one of the libraries below, the per-tool sections name the failure modes that drove Camelot's design choices, plus the kwargs you can reach for when Camelot's defaults don't fit your PDFs.

> **Note**
>
> This page was ported from the old GitHub wiki in 2026 and refreshed against current releases. Each per-tool section ends with a `Last verified: YYYY-MM-DD` footer — please open an issue if you find an entry has drifted out of date.

2.7.1 At a glance

Click any column header to sort. The Camelot column is highlighted; ✓ means "supported out of the box", "not supported", "partial / workaround required".

2.7.2 Side-by-side example

Picking a representative case — agstat.pdf, a ruled multi-row-header table from a US Department of Agriculture report — Camelot and Tabula both detect the table area, but Camelot picks up the merged-header row correctly without manual hinting:

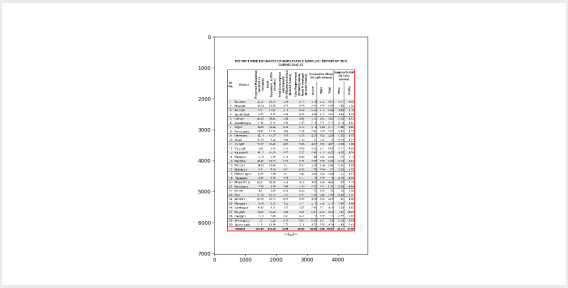


Fig. 1: Camelot `flavor='lattice'`, default kwargs.

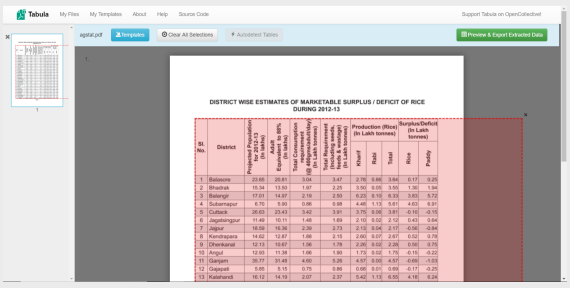


Fig. 2: Tabula auto-detect with the same PDF.

For a quick view of how each tool's CSV output differs on the same PDF, the docs/benchmark/ directory has per-tool CSVs alongside the source PDF for a dozen test cases (lattice + stream).

2.7.3 Tabula

Tabula is the most direct peer to Camelot — Camelot's `flavor='lattice'` / `'stream'` naming is in fact borrowed from Tabula. Tabula ships as a Java library plus a Python wrapper (tabula-py); the JVM dependency is the biggest difference for deployment.

- **When Tabula wins.** Auto-detection of stream-flavor tables is generally stronger than Camelot's *stream* parser — though Camelot's *network* and *hybrid* flavors (added in 1.0) close most of the gap on borderless tables. Tabula's interactive web UI for manually marking table regions is also unique.
- **When Camelot wins.** Multi-row column headers, merged spanning cells, and tables containing italic/superscript decorations. Camelot's `copy_text`, `shift_text`, `flag_size`, and `replace_text` kwargs let you fix specific extraction defects without leaving Python.

- **Deployment.** Camelot's pure-Python stack runs in any container that has `opencv-python-headless` and `pdfium`; Tabula needs a JRE.

Last verified: 2026-05-21 against tabula-java 1.0.5 / tabula-py 2.10.

2.7.4 pdfplumber

pdfplumber is a layout- analysis library that grew table-extraction features over time. It's built on *pdfminer.six* — the same backend Camelot used pre-2.0.

- **When pdfplumber wins.** When you want fine-grained access to *every* layout primitive (characters, rects, curves), not just the finished table. Pdfplumber exposes the raw layout objects directly, making it the right pick for "I want to find tables *and* the paragraph headers next to them".
- **When Camelot wins.** Out-of-the-box table-detection quality on the typical PDF report; per-table quality reports (`parsing_report` with `confidence`); the `flavor='hybrid'` parser combining lattice + network signals.
- **Backend.** Camelot has moved past pdfminer.six to playa-pdf for speed and encrypted-PDF correctness; pdfplumber still tracks pdfminer.six.

Last verified: 2026-05-21 against pdfplumber 0.11.5.

2.7.5 PyMuPDF (built-in tables)

PyMuPDF added a `Page.find_tables()` API in version 1.23 (2023). It's now a serious table-extractor backed by the C-level mupdf library.

- **When PyMuPDF wins.** Pure speed on simple ruled tables — rasterising is skipped entirely and the C parser is fast. Also a good pick if you're already using PyMuPDF for other PDF tasks (rendering, text search) and want to keep one dependency.
- **When Camelot wins.** Stream / network / hybrid flavors for borderless tables (PyMuPDF's table strategy is geometry-only); per-page parameter overrides; multi-page stitching helper.
- **License nuance.** PyMuPDF is AGPL — pulls open-source obligations into derivative work unless you buy a commercial licence. Camelot is MIT.

Last verified: 2026-05-21 against PyMuPDF 1.24.x.

2.7.6 gmft

gmft — "Give Me The Formatted Tables" — is a 2024-era tool that runs Microsoft's Table Transformer neural network for table detection plus structure recognition. A different shape from the rule-based tools above.

- **When gmft wins.** A pure model-first workflow on visually-complex tables — bank statements, forms — where you want the neural network to drive the whole extraction.
- **When Camelot wins.** Heuristic-first by default (predictable, CPU-only, no model weights) — and when you *do* want a model, Camelot's optional `flavor='ml'` runs the same Table Transformer family but fills cell **text from the PDF's own text layer** (or OCR for scans) instead of letting the model emit it, so it can't hallucinate or alter a value. Plus per-extraction kwargs and a per-table `confidence` score.
- **Resource cost.** gmft always pulls a Table Transformer checkpoint (~hundreds of MB) and benefits from a GPU. Camelot's core needs neither; that cost applies only if you opt into `camelot-py[ml]`.

Last verified: 2026-05-21 against gmft 0.4.x.

2.7.7 unstructured.io

unstructured is a document-preprocessing toolkit aimed at the LLM ingestion pipeline — it parses PDFs (plus DOCX, HTML, etc.) into a stream of typed elements (Title, NarrativeText, Table, …).

- **When unstructured wins.** Mixed-content documents where a table is one element among many and you want all of them in a single pipeline. The OCR / image fallback is built-in via plugins.
- **When Camelot wins.** Table-extraction-only workloads where you want maximum control over each table's parameters, want a per- table `confidence` score, or need the table as a pandas `DataFrame` rather than a Markdown / HTML serialisation.
- **Output.** unstructured returns tables as HTML / text snippets; Camelot returns pandas DataFrames + exporters for CSV / Excel / JSON / SQLite / Markdown.

Last verified: 2026-05-21 against unstructured 0.16.x.

2.7.8 tablers

tablers is a young, MIT-licensed extractor with its core algorithms written in **Rust** (exposed to Python via PyO3) and PDF handling through pdfium — so it installs with no external Python dependencies and is built for speed.

- **When tablers wins.** Raw speed on **ruled** tables — being Rust it is dramatically faster (see below), with lazy page loading for large files. If your PDFs are consistently ruled and throughput is the priority, it's worth a look.
- **When Camelot wins.** Extraction **quality** on ruled tables (numbers below), plus breadth Camelot has and tablers doesn't: borderless / whitespace tables (`stream` / `network` / `hybrid`), the optional neural `flavor="ml"` (incl. scanned PDFs), per-table `accuracy` / `whitespace` / `confidence` with `TableList.filter()`, multi-page stitching, and pandas-DataFrame output. tablers focuses on edge-detected tables and exports to CSV / Markdown / HTML.

Head-to-head on ruled tables

On the in-repo ICDAR-2013 set (67 born-digital, ruled-heavy PDFs), scored with Camelot's own metrics (`bench/benchmark_icdar.py` — an independent MIT implementation; the `TEDS` here is a difflib cell-text proxy, so read the columns relatively):

tool / config	F1	TEDS	row	col	time
camelot `lattice` (`engine="combined"`)	**0.778**	**0.789**	**0.762**	**0.829**	101 s
camelot `lattice` (`engine="vector"`)	0.766	0.784	0.748	0.806	13 s
tablers	0.750	0.724	0.657	0.741	1.5 s

So on ruled tables Camelot's lattice parser leads tablers on **every** quality metric — most notably row/col structure (row 0.762 vs 0.657, col 0.829 vs 0.741). tablers is the speed champion (Rust): ~67× faster than the `combined` engine here. Camelot's render-free `engine="vector"` narrows that to ~9× while keeping essentially all of combined's quality — a good middle ground when throughput matters but you still want Camelot's accuracy and breadth.

Last verified: 2026-05-25 against tablers 0.7.3, via the in-repo ``bench/benchmark_icdar.py`` harness.

2.7.9 Tools we no longer compare against

The earlier wiki page compared Camelot to two more tools that have since gone dormant; we evaluated current alternatives and dropped them from this page:

- pdftables — last release 2014, repository archived. Functional but unmaintained; no Python 3.10+ wheels.
- pdf-table-extract — last release 2017, dormant. Useful historical reference for the ruled-line / contour-detection approach but no active maintenance.

If either becomes active again, please open an issue and we'll add them back.

2.7.10 Keeping this page up-to-date

Each per-tool section ends with a `Last verified: YYYY-MM-DD` marker so drift is visible without having to dig through commit history. The intent is for one of these to fall out of date — that's expected — and for a contributor to refresh it via PR when they notice. The per-tool prose + capability matrix above are hand-maintained.

The objective numbers — does each tool run on a given PDF, how many tables it returns, and how long it takes — are produced by a script, so they can be refreshed without editing prose:

```
$ python bench/comparison.py
```

That runs Camelot plus every peer extractor that's importable in the environment (missing ones are skipped, not errored) against a small canonical corpus, and writes `docs/_static/comparison_bench.csv`. The script measures table-count + timing only, not extraction *quality* (which needs per-PDF ground truth — a separate effort). Wiring it into a release-time CI job that installs the heavyweight comparators (a JRE for Tabula, PyTorch for gmft, ...) so the CSV refreshes automatically is the remaining follow-up.

For practical recipes that *use* Camelot's specific features — `per_page`, `replace_text`, in-memory `bytes` input, `stack_contiguous` for multi-page tables — see the *advanced* page.

2.8 Migrating to 2.0

Camelot 2.0 rolls up a backend migration, performance work, an optional neural backend, and a few small breaking changes. For most users `import camelot` and `camelot.read_pdf(...)` keep working unchanged. This page lists what to check when upgrading from 1.0.x, then points at the new opt-in features.

The full, dated list of changes is in the changelog.

2.8.1 Breaking changes

Python 3.10+

Python 3.9 (EOL October 2025) is no longer supported. The minimum is now **Python 3.10**.

`line_scale` default is 15 (was documented as 40)

The CLI and `read_pdf` docstring used to *say* the `flavor="lattice"` default `line_scale` was 40, but the Lattice parser always defaulted to 15. The docs now match the implementation. If you relied on the documented-but-unimplemented 40, set it explicitly:

```
camelot.read_pdf("file.pdf", flavor="lattice", line_scale=40)
```

`Table.to_excel` drops the index/header by default

`Table.to_excel` now defaults to `index=False, header=False` to match `Table.to_csv` — Excel exports no longer carry the pandas auto-generated row index / column header. Opt back in with:

```
table.to_excel("out.xlsx", index=True, header=True)
```

`TableList` materialises its input

`TableList(...)` now consumes an iterable into a list at construction (so `bool()` / `len()` work on `TableList(generator())`). A generator passed in is exhausted immediately rather than at first access.

`PDFHandler.pages` is a property

`PDFHandler.pages` is now a lazily-resolved property (was an attribute). Reads are unchanged; only code that *set* it after subclassing is affected.

PDF backend is now `playa-pdf`

The backend moved from `pypdf` + `pdfminer.six` to playa-pdf: a smaller install set, more accurate encrypted-PDF handling, and faster hot paths. Pure `import camelot` callers should see no API change. `pdfminer.six` is no longer a direct dependency — `playa.miner` exposes a PDFMiner-compatible layout API, so imports through Camelot keep working.

Default lattice `engine` is `"combined"`

`flavor="lattice"` now defaults to `engine="combined"` (raster OpenCV detection **plus** the PDF's native vector ruled lines). It is safe by construction — raster always runs and vector lines can only add — so it is never worse than the old `"raster"` default. Pass `engine="raster"` to restore the exact pre-2.0 behaviour. (There is no `engine="auto"`.)

2.8.2 New (opt-in) features

Neural backend for borderless / scanned tables

A new optional `flavor="ml"` uses a Table Transformer model for table **structure** and fills cell **text** from the PDF (so it can't hallucinate values). It targets borderless tables, where the heuristic parsers plateau. With OCR it also reads **scanned / image-only** PDFs. These pull heavier dependencies, imported lazily, so the core install is unaffected:

```
pip install "camelot-py[ml]"       # borderless
pip install "camelot-py[ml,ocr]"   # + scanned PDFs
```

```
tables = camelot.read_pdf("report.pdf", flavor="ml")  # device="cuda"/"xpu" optional
```

See *How It Works* for the design and a borderless benchmark, and *How Camelot compares to other tools* for how Camelot's flavors line up against other tools.

Other additions worth knowing

- `flavor="auto"` picks `lattice` or `network` per page.
- `TableList.filter(...)` drops low-quality tables by row/column count, accuracy, or whitespace.
- `Table.confidence` — a unified `[0, 1]` quality score in `parsing_report`.
- `per_page=` overrides, `replace_text=`, list-form `strip_text=`, `bytes` / file-like `read_pdf` input, and a `cpu_count` cap for parallel runs.

2.9 Frequently Asked Questions

This part of the documentation answers some common questions. To add questions, please open an issue here.

2.9.1 Does Camelot work with image-based PDFs?

No, Camelot only works with text-based PDFs and not scanned documents. (As Tabula explains, "If you can click and drag to select text in your table in a PDF viewer, then your PDF is text-based".)

2.9.2 How to reduce memory usage for long PDFs?

When extracting tables from a long PDF in one call, RAM grows roughly with the number of pages held in memory at once: every page's text objects, the per-parser caches, and the resulting `TableList` all live until `read_pdf` returns.

The simplest mitigation is to process the document in page-range chunks, write each chunk's tables to disk, and let Python free the intermediate state between calls. The pattern below uses only the public API and works on any range of pages — concept originally from #90 by @nightwarriorftw and @anakin87:

```
import camelot


def extract_in_chunks(
    filepath,
    total_pages,
    chunk_size=50,
    export_dir=".",
    **read_pdf_kwargs,
):
    """Extract tables a chunk of pages at a time, freeing RAM between chunks.

    Parameters
    ----------
    filepath : str
        Path to the PDF file.
    total_pages : int
        Total page count of the PDF. Get it from any PDF tool, e.g.
        ``len(playa.parse(open(filepath, "rb").read()).pages)``.
    chunk_size : int, optional (default: 50)
        How many pages to process per ``read_pdf`` call.
    export_dir : str, optional (default: ".")
        Directory in which per-chunk CSVs are written.
    **read_pdf_kwargs
        Any other keyword arguments are forwarded to
        :meth:`camelot.read_pdf` (e.g. ``flavor="stream"``,
        ``table_areas=...``).
    """
    for start in range(1, total_pages + 1, chunk_size):
        end = min(start + chunk_size - 1, total_pages)
        tables = camelot.read_pdf(
            filepath, pages=f"{start}-{end}", **read_pdf_kwargs
        )
        tables.export(f"{export_dir}/tables_{start}-{end}.csv")
```

Each iteration's `TableList` becomes unreachable after the `tables.export(...)` call, so the per-chunk PDF parse state is released before the next chunk runs. For very long documents, combine this with `flavor="stream"` or `flavor="network"` (cheaper than Lattice's image conversion) where the table layouts allow it.

2.9.3 How can I supply my own image conversion backend to Lattice?

When using the *Lattice* flavor, you can supply your own *image conversion backend* by creating a class with a `convert` method as follows:

```
>>> class ConversionBackend(object):
>>>     def convert(pdf_path, png_path):
>>>         # read pdf page from pdf_path
>>>         # convert pdf page to image
>>>         # write image to png_path
>>>         pass
>>>
>>> tables = camelot.read_pdf(filename, backend=ConversionBackend())
```

2.9.4 Why don't a table's bbox coordinates line up with the page image?

A `Table`'s `_bbox` (and the coordinates in its cells) live in **PDF coordinate space**, while `table.get_pdf_image()` returns the page as a rendered raster in **image coordinate space**. The two differ in two ways, so drawing `_bbox` straight onto the image puts the box in the wrong place:

- **Origin / direction.** PDF space has its origin at the *bottom-left* with y increasing *upward*; image space has its origin at the *top-left* with y increasing *downward*. The y-axis must be flipped.
- **Scale.** PDF coordinates are in points (1/72 inch). The image is rendered at a higher resolution (300 dpi by default), so it is roughly `300/72` times larger on each axis.

`Table` exposes the page size as `table.pdf_size (width, height)`, which together with the rendered image's pixel size gives the per-axis scale. To overlay a table's bbox on its page image:

```
import camelot
import cv2

tables = camelot.read_pdf("foo.pdf", flavor="lattice")
table = tables[0]

img = table.get_pdf_image()          # rendered raster (BGR)
image_h, image_w = img.shape[:2]
pdf_w, pdf_h = table.pdf_size
scale_x, scale_y = image_w / pdf_w, image_h / pdf_h

x0, y0, x1, y1 = table._bbox         # PDF coords (origin bottom-left)
top_left = (round(x0 * scale_x), round((pdf_h - y1) * scale_y))
bottom_right = (round(x1 * scale_x), round((pdf_h - y0) * scale_y))

cv2.rectangle(img, top_left, bottom_right, (0, 255, 0), 3)
cv2.imwrite("foo_bbox.jpg", img)
```

The same scale-and-flip converts any PDF-space coordinate (a cell, a joint) into the image, and inverting it converts an image-space coordinate back into PDF space.

2.9.5 I have table coordinates from an image — how do I pass them to `table_areas`?

A common workflow is to detect a table's region with an external, image-based tool (an ML layout detector such as table-transformers, or manual annotation on a rendered page) and then have Camelot extract just that region via `table_areas`. Because `table_areas` is in **PDF coordinate space** (origin bottom-left, points), you have to convert your **image** coordinates the other way — the inverse of the section above.

The key detail: use the DPI of *your own* render. If you rasterised the page yourself (e.g. with `pdf2image` at `dpi=D`), then `image_width = pdf_width * D / 72`, so the per-axis scale back to PDF points is `72 / D` — **not** Camelot's internal 300 dpi.

```
import camelot

# Box from an image-based detector: (left, top, right, bottom) in
# pixels, origin top-left, on a page you rendered at `dpi`.
x0_img, y0_img, x1_img, y1_img = detected_box
dpi = 200  # whatever you passed to pdf2image / your renderer

# image px -> PDF points
s = 72.0 / dpi
pdf_w_x0, pdf_w_x1 = x0_img * s, x1_img * s
# flip y: image top-left origin -> PDF bottom-left origin.
# page height in points = image_height_px * 72 / dpi
page_h_pts = image_height_px * s
pdf_top = page_h_pts - y0_img * s        # image top  -> larger PDF y
pdf_bottom = page_h_pts - y1_img * s     # image bot  -> smaller PDF y

# table_areas wants "x1,y1,x2,y2" = top-left, bottom-right (PDF space)
area = f"{pdf_w_x0},{pdf_top},{pdf_w_x1},{pdf_bottom}"
tables = camelot.read_pdf("doc.pdf", flavor="lattice", table_areas=[area])
```

If your detector ran on an image Camelot itself produced (rather than your own `pdf2image` render), use `table.pdf_size` and the rendered image size to get the scale, exactly as in the previous answer but inverted.

2.9.6 I get `AttributeError: module 'camelot' has no attribute 'read_pdf'`

This almost always means the **wrong package** is installed. There is an unrelated project on PyPI also called `camelot` (a configuration library); `pip install camelot` installs *that*, not this table- extraction library. Uninstall it and install `camelot-py`:

```
$ pip uninstall camelot camelot-py
$ pip install "camelot-py[base]"
```

The import name is still `camelot` (`import camelot`) — only the *install* name differs (`camelot-py`).

If the right package is installed and you still hit this, check that you don't have a local file named `camelot.py` (or a `camelot/` folder) in your working directory shadowing the installed package — rename it and try again.

2.10 Command-Line Interface

Camelot comes with a command-line interface.

You can print the help for the interface by typing `camelot --help` in your favorite terminal program, as shown below. Furthermore, you can print the help for each command by typing `camelot <command> --help`. Try it out!

2.10.1 Running without installing (uvx)

If you only want to use the CLI ad-hoc, uvx lets you run it without installing Camelot into the current environment:

```
$ uvx camelot-py lattice --output tables.csv document.pdf
```

The `camelot-py` console script is an alias for `camelot` matching the PyPI package name, so the older `uvx --from camelot-py camelot ...` invocation also still works.

2.10.2 Format inference

`--format` is optional — when omitted, Camelot infers the format from the `--output` path's extension. Supported extensions:

So this works:

```
$ camelot-py lattice --output tables.xlsx document.pdf
# equivalent to: camelot-py lattice --format excel --output tables.xlsx document.pdf
```

2.10.3 Output is a template

`--output` is treated as a *template* — each detected table is written to `<output_stem>-page-<P>-table-<T>.<ext>`. So `--output report.csv` on a document with 2 tables on page 1 and 1 table on page 3 produces `report-page-1-table-1.csv`, `report-page-1-table-2.csv`, `report-page-3-table-1.csv`.

camelot

Camelot: PDF Table Extraction for Humans.

Usage

```
camelot [OPTIONS] COMMAND [ARGS]...
```

Options

--version

Show the version and exit.

Commands

hybrid

Combines the strengths of both the Network…

lattice

Use lines between text to parse the table.

network

Use text alignments to parse the table.

stream

Use spaces between text to parse the table.

CHAPTER

THREE

THE API DOCUMENTATION/GUIDE

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

3.1 API Reference

3.1.1 Main Interface

camelot.**read_pdf**(*filepath: str | Path | bytes | bytearray | memoryview | IO[bytes]*, *pages='1'*, *password=None*, *flavor='lattice'*, *suppress_stdout=False*, *parallel=False*, *cpu_count=None*, *layout_kwargs=None*, *per_page=None*, *debug=False*, ***kwargs*)

Read PDF and return extracted tables.

Note: kwargs annotated with ^ can only be used with flavor='stream' or flavor='network' and kwargs annotated with * can only be used with flavor='lattice'. The hybrid parser accepts kwargs with both annotations.

Parameters

- **filepath** (*str, Path, bytes, or binary file-like*) – Source PDF. Accepts a filesystem path / URL, a `bytes`-like object, or any binary stream with a `.read()` method (`io.BytesIO`, an open `"rb"` file, `requests` response `.raw`, etc). For in-memory inputs the bytes are spilled to a temporary file once and cleaned up on context-manager exit, so the Lattice OpenCV image-conversion backend keeps working unchanged. Originally requested in #170 / #245 / #270.
- **pages** (*str, optional (default: '1')*) – Comma-separated page numbers. Example: '1,3,4' or '1,4-end' or 'all'.
- **password** (*str, optional (default: None)*) – Password for decryption.
- **flavor** (*str (default: 'lattice')*) – The parsing method to use. Valid values:
  - `'lattice'` (default): line-ruled tables.
  - `'stream'`: borderless tables with whitespace-separated columns.
  - `'network'`: borderless tables via text-edge alignment connectivity.
  - `'hybrid'`: combines layout- and image-based analysis.
  - `'ml'`: neural table-structure recognition (Table Transformer) for the structure, with cell text filled from the PDF's own text layer (no hallucinated values). Requires the optional ML dependencies: `pip install 'camelot-py[ml]'`. Best for borderless tables where the heuristic parsers plateau.
  - `'auto'`: detect the flavor **per page** (count ruled lines on each rendered page) and parse each group accordingly — ruled pages via `lattice` with `engine='combined'`, the rest via `network` — then merge. Handles documents that mix text-only cover pages with ruled

tables deeper in. A `UserWarning` reports the per-page choices. (More accurate but slower, since it renders every page for the probe.)

- **suppress_stdout** (*bool, optional (default: False)*) – Print all logs and warnings.
- **parallel** (*bool, optional (default: False)*) – Process pages in parallel using all available cpu cores.
- **cpu_count** (*int, optional (default: None)*) – Maximum number of worker processes when `parallel=True`. `None` (default) uses all available cores. Values are clamped to `[1, multiprocessing.cpu_count()]`. Ignored when `parallel=False`.
- **layout_kwargs** (*dict, optional (default: {})*) – A dict of pdfminer.layout.LAParams kwargs.
- **per_page** (*dict, optional (default: None)*) – Per-page parameter overrides. Maps a 1-indexed page number (int or str) to a dict of any keyword argument otherwise valid for `read_pdf`. Values supplied here override the globally-supplied kwargs for that one page only — every other page keeps the global values. Useful for multi-layout PDFs where different pages need different `table_areas`, `columns`, `flavor`, etc. The per-page `flavor` itself may be overridden; the global flavor applies otherwise. Originally proposed by @sverma25 in #41.

  Example:

```
tables = camelot.read_pdf(
    "report.pdf",
    pages="1-3",
    flavor="stream",
    split_text=True,
    per_page={2: {"table_areas": ["120, 210, 400, 90"]}},
)
```

  Here pages 1 and 3 use the global `flavor="stream", split_text=True` only; page 2 uses both *and* the page-specific `table_areas`.
- **table_areas** (*list, optional (default: None)*) – List of table area strings of the form x1,y1,x2,y2 where (x1, y1) -> left-top and (x2, y2) -> right-bottom in PDF coordinate space.
- **columns^** (*list, optional (default: None)*) – List of column x-coordinates strings where the coordinates are comma-separated.
- **split_text** (*bool, optional (default: False)*) – Split text that spans across multiple cells.
- **flag_size** (*bool, optional (default: False)*) – Flag text based on font size. Useful to detect super/subscripts. Adds <s></s> around flagged text.
- **strip_text** (*str or sequence of str, optional (default: "")*) – Characters or substrings to strip from each cell before assignment. A `str` strips per-character — every character in the string is removed wherever it appears (e.g. `" \n"` drops all spaces and newlines). A list/tuple of `str` strips whole substrings (e.g. `["[1]", "[2]"]` removes those footnote markers but leaves bare `[/]` alone). Whole-substring mode requested in #484.
- **replace_text** (*dict, optional (default: None)*) – Mapping of substring → replacement applied to every cell's text just before it is written into the table. Keys are matched as literal substrings (regex metacharacters are escaped). Useful for collapsing soft-broken words (e.g. `{" \n": " "}`), normalising abbreviations, or rewriting unit names. Distinct

from `strip_text` which can only remove characters; this can replace with arbitrary text. Requested in #481. (#482)

- **row_tol^** (*int, optional (default: 2)*) – Tolerance parameter used to combine text vertically, to generate rows.
- **column_tol^** (*int, optional (default: 0)*) – Tolerance parameter used to combine text horizontally, to generate columns.
- **process_background*** (*bool, optional (default: False)*) – Process background lines.
- **line_scale*** (*int, optional (default: 15)*) – Line size scaling factor. The larger the value the smaller the detected lines. Making it very large will lead to text being detected as lines.
- **copy_text*** (*list, optional (default: None)*) – {'h', 'v'} Direction in which text in a spanning cell will be copied over.
- **shift_text*** (*list, optional (default: ['l', 't'])*) – {'l', 'r', 't', 'b'} Direction in which text in a spanning cell will flow.
- **line_tol*** (*int, optional (default: 2)*) – Tolerance parameter used to merge close vertical and horizontal lines.
- **joint_tol*** (*int, optional (default: 2)*) – Tolerance parameter used to decide whether the detected lines and points lie close to each other.
- **threshold_blocksize*** (*int, optional (default: 15)*) – Size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on.

  For more information, refer OpenCV's adaptiveThreshold.
- **threshold_constant*** (*int, optional (default: -2)*) – Constant subtracted from the mean or weighted mean. Normally, it is positive but may be zero or negative as well.

  For more information, refer OpenCV's adaptiveThreshold.
- **iterations*** (*int, optional (default: 0)*) – Number of dilation passes applied to close small gaps in the line mask.

  For more information, refer OpenCV's dilate.
- **erode_iterations*** (*int, optional (default: 0)*) – Number of erosion passes applied **after** dilation. Set equal to `iterations` for a morphological closing — bridges gaps in ruled lines without thickening the mask overall (which avoids the spurious extra-row artefact reported in #363). (#363)
- **backend*** (*str, optional by default "pdfium"*) – The backend to use for converting the PDF to an image so it can be processed by OpenCV.
- **use_fallback*** (*bool, optional*) – Fallback to another backend if unavailable, by default True
- **resolution*** (*int, optional (default: 300)*) – Resolution used for PDF to PNG conversion.
- **engine*** (*str, optional (default: 'combined')*) – Line-detection engine for `flavor='lattice'` (and the lattice half of `flavor='hybrid'`):
  - `'combined'` (default): render the page and detect ruled lines with OpenCV **and** union in the ruled lines read from the PDF's native vector graphics, so tables whose rules render faintly (vector strokes, anti-aliasing) are still found. Safe by construction — raster always

runs, vector lines can only add, and they're clipped to `table_regions` — so it never does worse than `'raster'` (#763).

- `'raster'`: render the page and detect ruled lines with OpenCV only — the pre-#763 behaviour.
- `'vector'`: detect tables straight from the PDF's vector ruled lines, skipping rasterisation entirely — the fastest path, for PDFs whose tables are drawn with real vector strokes (#763).

With `flavor='hybrid'` the same choices select how its lattice half finds ruled lines; `engine='vector'` there is the **render-free hybrid** — vector ruled lines merged with the network text-edge alignment — for partial-ruled / borderless tables at roughly an order of magnitude less time than the raster path (#39).

Returns
tables

Return type
camelot.core.TableList

Notes

Encrypted PDFs / extraction permissions (#590). Camelot honours the `/Encrypt` dictionary's text-extraction permission: `read_pdf` raises `playa.exceptions.PDFTextExtractionNotAllowed` if the PDF is encrypted and the user-password permission set forbids text extraction. The check fires on the document object returned by `playa.open` while the encryption metadata is still attached — this is a real behavioural change vs the pre-1.0 backend, where per-page temp-PDF splitting silently dropped the metadata so the check was effectively a no-op. Note: PDF spec only enforces the flag through the encryption layer — for **unencrypted** PDFs that carry a "no extraction" claim via `/Perms`, there is no enforcement mechanism and Camelot extracts. Supplying the document owner password through `password=` bypasses the user-password permission set (matches every other PDF tool).

Examples

```
>>> import camelot
>>> tables = camelot.read_pdf("foo.pdf")  # xdoctest: +SKIP
>>> tables.n  # xdoctest: +SKIP
1
>>> tables[0].df  # xdoctest: +SKIP
>>> tables[0].to_csv("foo.csv")  # xdoctest: +SKIP
```

Select a parser and restrict extraction to a page range:

```
>>> tables = camelot.read_pdf(  # xdoctest: +SKIP
...     "foo.pdf", flavor="lattice", pages="1-3"
... )
```

3.1.2 Lower-Level Classes

class camelot.handlers.**PDFHandler**(*filepath: str | Path | bytes | bytearray | memoryview | IO[bytes]*, *pages='1'*, *password=None*, *debug=False*)

Handles all operations on the PDF's.

Handles all operations like temp directory creation, splitting file into single page PDFs, parsing each PDF and then removing the temp directory.

Parameters

- **filepath** (*str, Path, bytes, or binary file-like*) – Source PDF. Accepts a filesystem path / URL, or — since #270 — a `bytes`-like object or any binary stream with a `.read()` method (`io.BytesIO`, an open `"rb"` file, `requests` response `.raw`, etc). In the in-memory cases the bytes are spilled to a temporary file once and cleaned up when the handler is closed; this keeps the rest of the pipeline (in particular the Lattice OpenCV image-conversion backend) unchanged.
- **pages** (*str, optional (default: '1')*) – Comma-separated page numbers. Example: '1,3,4' or '1,4-end' or 'all'.
- **password** (*str, optional (default: None)*) – Password for decryption.
- **debug** (*bool, optional (default: False)*) – Whether the parser should store debug information during parsing.

`close()` → None

Delete the URL-downloaded temp file, if any.

Idempotent; safe to call from both `__exit__` and an explicit `handler.close()` call. No-op when `filepath` was a user-owned path (we never delete a file the caller passed in).

`property pages: list[int]`

Resolved 1-based page numbers, sorted and de-duplicated.

Lazy: only opens the PDF if the spec is something other than the default `"1"`. Cached after first access.

parse(*flavor: str = 'lattice', suppress_stdout: bool = False, parallel: bool = False, cpu_count: int | None = None, layout_kwargs: dict[str, Any] | None = None, per_page: dict[int, dict[str, Any]] | None = None, pages: list[int] | None = None, render_cache: dict[int, str] | None = None, **kwargs*)

Extract tables by calling parser.get_tables on all single page PDFs.

Parameters

- **flavor** (*str (default: 'lattice')*) – The parsing method to use. Lattice is used by default.
- **suppress_stdout** (*bool (default: False)*) – Suppress logs and warnings.
- **parallel** (*bool (default: False)*) – Process pages in parallel using all available cpu cores.
- **cpu_count** (*int, optional (default: None)*) – Maximum number of worker processes to use when `parallel` is True. `None` (default) uses all available cores. Values are clamped to `[1, multiprocessing.cpu_count()]`. Ignored when `parallel` is False.
- **layout_kwargs** (*dict, optional (default: {})*) – A dict of pdfminer.layout.LAParams kwargs.
- **kwargs** (*dict*) – See camelot.read_pdf kwargs.

Returns

tables – List of tables found in PDF.

Return type

camelot.core.TableList

`class camelot.parsers.`**Stream**(*table_regions=None, table_areas=None, columns=None, split_text=False, flag_size=False, strip_text='', replace_text=None, edge_tol=50, row_tol=2, column_tol=0, **kwargs*)

Stream method of parsing looks for spaces between text to parse the table.

If you want to specify columns when specifying multiple table areas, make sure that the length of both lists are equal.

Parameters

- **table_regions** (*list, optional (default: None)*) – List of page regions that may contain tables of the form x1,y1,x2,y2 where (x1, y1) -> left-top and (x2, y2) -> right-bottom in PDF coordinate space.
- **table_areas** (*list, optional (default: None)*) – List of table area strings of the form x1,y1,x2,y2 where (x1, y1) -> left-top and (x2, y2) -> right-bottom in PDF coordinate space.
- **columns** (*list, optional (default: None)*) – List of column x-coordinates strings where the coordinates are comma-separated.
- **split_text** (*bool, optional (default: False)*) – Split text that spans across multiple cells.
- **flag_size** (*bool, optional (default: False)*) – Flag text based on font size. Useful to detect super/subscripts. Adds <s></s> around flagged text.
- **strip_text** (*str, optional (default: '')*) – Characters that should be stripped from a string before assigning it to a cell.
- **edge_tol** (*int, optional (default: 50)*) – Tolerance parameter for extending textedges vertically.
- **row_tol** (*int, optional (default: 2)*) – Tolerance parameter used to combine text vertically, to generate rows.
- **column_tol** (*int, optional (default: 0)*) – Tolerance parameter used to combine text horizontally, to generate columns.

compute_parse_errors(*table*)

Compute parse errors for the table .

Parameters
table (camelot.core.Table)

Returns
Parse errors

Return type
Tuple

extract_tables()

Extract tables from the document.

prepare_page_parse(*filename*, *layout*, *dimensions*, *page_idx*, *images*, *horizontal_text*, *vertical_text*, *rotation*, *layout_kwargs*)

Prepare the page for parsing.

record_parse_metadata(*table*)

Record data about the origin of the table.

table_bboxes()

Return a list of table bounding boxes sorted by position .

Returns
[description]

Return type
[type]

class camelot.parsers.**Lattice**(*table_regions=None*, *table_areas=None*, *process_background=False*, *line_scale=15*, *copy_text=None*, *shift_text=None*, *split_text=False*, *flag_size=False*, *strip_text=''*, *replace_text=None*, *line_tol=2*, *joint_tol=2*, *threshold_blocksize=15*, *threshold_constant=-2*, *iterations=0*, *erode_iterations=0*, *resolution=300*, *use_fallback=True*, *backend='pdfium'*, *engine='combined'*, ***kwargs*)

Lattice method looks for lines between text to parse the table.

Parameters

- **table_regions** (*list, optional (default: None)*) – List of page regions that may contain tables of the form x1,y1,x2,y2 where (x1, y1) -> left-top and (x2, y2) -> right-bottom in PDF coordinate space.
- **table_areas** (*list, optional (default: None)*) – List of table area strings of the form x1,y1,x2,y2 where (x1, y1) -> left-top and (x2, y2) -> right-bottom in PDF coordinate space.
- **process_background** (*bool, optional (default: False)*) – Process background lines.
- **line_scale** (*int, optional (default: 15)*) – Line size scaling factor. The larger the value the smaller the detected lines. Making it very large will lead to text being detected as lines.
- **copy_text** (*list, optional (default: None)*) – {'h', 'v'} Direction in which text in a spanning cell will be copied over.
- **shift_text** (*list, optional (default: ['l', 't'])*) – {'l', 'r', 't', 'b'} Direction in which text in a spanning cell will flow.
- **split_text** (*bool, optional (default: False)*) – Split text that spans across multiple cells.
- **flag_size** (*bool, optional (default: False)*) – Flag text based on font size. Useful to detect super/subscripts. Adds <s></s> around flagged text.
- **strip_text** (*str, optional (default: '')*) – Characters that should be stripped from a string before assigning it to a cell.
- **line_tol** (*int, optional (default: 2)*) – Tolerance parameter used to merge close vertical and horizontal lines.
- **joint_tol** (*int, optional (default: 2)*) – Tolerance parameter used to decide whether the detected lines and points lie close to each other.
- **threshold_blocksize** (*int, optional (default: 15)*) – Size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on.

  For more information, refer OpenCV's adaptiveThreshold.
- **threshold_constant** (*int, optional (default: -2)*) – Constant subtracted from the mean or weighted mean. Normally, it is positive but may be zero or negative as well.

  For more information, refer OpenCV's adaptiveThreshold.

- **iterations** (*int, optional (default: 0)*) – Number of dilation passes applied to close small gaps in the line mask (useful when a table's ruled lines don't quite meet at corners).

  For more information, refer OpenCV's dilate.

- **erode_iterations** (*int, optional (default: 0)*) – Number of erosion passes applied **after** dilation. Set equal to `iterations` for a morphological closing (bridges gaps without thickening the mask, which avoids spurious extra rows above/below the detected table). See #363.

- **backend*** (*str, optional by default "pdfium"*) – The backend to use for converting the PDF to an image so it can be processed by OpenCV.

- **use_fallback*** (*bool, optional*) – Fallback to another backend if unavailable, by default True

- **resolution** (*int, optional (default: 300)*) – Resolution used for PDF to PNG conversion.

- **engine** (*str, optional (default: 'combined')*) – Line-detection engine (lattice only):

  - `'combined'` (default): OpenCV on the rendered page **plus** the PDF's native vector ruled lines unioned into the line masks before contour/joint detection — recovers tables whose rules render faintly. Safe by construction (raster always runs first, vector lines can only add; vector lines are clipped to `table_regions` so it never expands a table past the region).
  - `'raster'`: OpenCV on the rendered page only (the pre-#763 behaviour).
  - `'vector'`: detect tables purely from the PDF's vector ruled lines, skipping rasterisation entirely — fastest, for PDFs whose tables are drawn with real vector strokes (#763).

compute_parse_errors(*table*)

Compute parse errors for the table .

Parameters
table (camelot.core.Table)

Returns
Parse errors

Return type
Tuple

extract_tables()

Extract tables from the document.

prepare_page_parse(*filename*, *layout*, *dimensions*, *page_idx*, *images*, *horizontal_text*, *vertical_text*, *rotation*, *layout_kwargs*)

Prepare the page for parsing.

record_parse_metadata(*table*)

Record data about the origin of the table.

table_bboxes()

Return a list of table bounding boxes sorted by position .

Returns
[description]

Return type
[type]

class camelot.parsers.**Network**(*table_regions=None*, *table_areas=None*, *columns=None*, *flag_size=False*, *split_text=False*, *strip_text=''*, *replace_text=None*, *edge_tol=None*, *row_tol=2*, *column_tol=0*, *debug=False*, ***kwargs*)

Network method looks for spaces between text to parse the table.

If you want to specify columns when specifying multiple table areas, make sure that the length of both lists are equal.

Parameters

- **table_regions** (*list, optional (default: None)*) – List of page regions that may contain tables of the form x1,y1,x2,y2 where (x1, y1) -> left-top and (x2, y2) -> right-bottom in PDF coordinate space.
- **table_areas** (*list, optional (default: None)*) – List of table area strings of the form x1,y1,x2,y2 where (x1, y1) -> left-top and (x2, y2) -> right-bottom in PDF coordinate space.
- **columns** (*list, optional (default: None)*) – List of column x-coordinates strings where the coordinates are comma-separated.
- **split_text** (*bool, optional (default: False)*) – Split text that spans across multiple cells.
- **flag_size** (*bool, optional (default: False)*) – Flag text based on font size. Useful to detect super/subscripts. Adds <s></s> around flagged text.
- **strip_text** (*str, optional (default: '')*) – Characters that should be stripped from a string before assigning it to a cell.
- **edge_tol** (*int, optional (default: 50)*) – Tolerance parameter for extending textedges vertically.
- **row_tol** (*int, optional (default: 2)*) – Tolerance parameter used to combine text vertically, to generate rows.
- **column_tol** (*int, optional (default: 0)*) – Tolerance parameter used to combine text horizontally, to generate columns.

compute_parse_errors(*table*)

Compute parse errors for the table .

Parameters
table (camelot.core.Table)

Returns
Parse errors

Return type
Tuple

extract_tables()

Extract tables from the document.

prepare_page_parse(*filename*, *layout*, *dimensions*, *page_idx*, *images*, *horizontal_text*, *vertical_text*, *rotation*, *layout_kwargs*)

Prepare the page for parsing.

record_parse_metadata(*table*)

Record data about the origin of the table.

table_bboxes()

Return a list of table bounding boxes sorted by position .

Returns
[description]

Return type
[type]

class camelot.parsers.**Hybrid**(*table_regions=None*, *table_areas=None*, *columns=None*, *flag_size=False*, *split_text=False*, *strip_text=''*, *replace_text=None*, *edge_tol=None*, *row_tol=2*, *column_tol=0*, *debug=False*, *engine='combined'*, ***kwargs*)

Defines a hybrid parser, leveraging both network and lattice parsers.

Parameters

- **table_regions** (*list, optional (default: None)*) – List of page regions that may contain tables of the form x1,y1,x2,y2 where (x1, y1) -> left-top and (x2, y2) -> right-bottom in PDF coordinate space.
- **table_areas** (*list, optional (default: None)*) – List of table area strings of the form x1,y1,x2,y2 where (x1, y1) -> left-top and (x2, y2) -> right-bottom in PDF coordinate space.
- **columns** (*list, optional (default: None)*) – List of column x-coordinates strings where the coordinates are comma-separated.
- **split_text** (*bool, optional (default: False)*) – Split text that spans across multiple cells.
- **flag_size** (*bool, optional (default: False)*) – Flag text based on font size. Useful to detect super/subscripts. Adds <s></s> around flagged text.
- **strip_text** (*str or sequence of str, optional (default: '')*) – Characters or substrings to strip from each cell. A `str` strips per-character; a list/tuple of `str` strips whole substrings (#484).
- **edge_tol** (*int, optional (default: 50)*) – Tolerance parameter for extending textedges vertically.
- **row_tol** (*int, optional (default: 2)*) – Tolerance parameter used to combine text vertically, to generate rows.
- **column_tol** (*int, optional (default: 0)*) – Tolerance parameter used to combine text horizontally, to generate columns.
- **engine** (*str, optional (default: 'combined')*) – Line-detection engine for hybrid's **lattice half** (the network half is text-based and unaffected):
  - `'combined'` (default): OpenCV on the rendered page **plus** the PDF's native vector ruled lines unioned in — recovers faintly-rendered rules. Matches the `flavor='lattice'` default.
  - `'raster'`: detect ruled lines with OpenCV only (pre-#763).
  - `'vector'`: detect ruled lines **straight from the PDF's vector graphics, skipping rasterisation and OpenCV entirely** — the render-free hybrid (network text-edge alignment merged with vector ruled lines) for partial-ruled / borderless tables at roughly an order of magnitude less time than the raster path. (#39)

compute_parse_errors(*table*)

Compute parse errors for the table .

Parameters
table (camelot.core.Table)

Returns
Parse errors

Return type
Tuple

extract_tables()

Extract tables from the document.

prepare_page_parse(*filename*, *layout*, *dimensions*, *page_idx*, *images*, *horizontal_text*, *vertical_text*, *rotation*, *layout_kwargs*)

Call this method to prepare the page parsing .

Parameters

- **filename** (*[type]*) – [description]
- **layout** (*[type]*) – [description]
- **dimensions** (*[type]*) – [description]
- **page_idx** (*[type]*) – [description]
- **layout_kwargs** (*[type]*) – [description]

record_parse_metadata(*table*)

Record data about the origin of the table.

table_bboxes()

Return a list of table bounding boxes sorted by position .

Returns
[description]

Return type
[type]

3.1.3 Lower-Lower-Level Classes

class camelot.core.**TableList**(*tables: Iterable[*Table*]*)

Defines a list of camelot.core.Table objects.

Each table can be accessed using its index.

n

Number of tables in the list.

Type
int

Examples

```
>>> from camelot.core import TableList
>>> tables = TableList([])
>>> tables.n
0
>>> tables
<TableList n=0>
```

export(*path: str*, *f='csv'*, *compress=False*)

Export the list of tables to specified file format.

Parameters

- **path** (*str*) – Output filepath.
- **f** (*str*) – File format. Can be csv, excel, html, json, markdown or sqlite.
- **compress** (*bool*) – Whether or not to add files to a ZIP archive.

filter(*min_rows: int = 1*, *min_columns: int = 1*, *min_accuracy: float = 0.0*, *max_whitespace: float = 100.0*) → *TableList*

Return a new TableList keeping only tables that pass all thresholds.

A post-extraction convenience for dropping noise / low-quality tables (single stray cells, mostly-empty regions, …). Parsing is unchanged — everything is still detected; this just selects from the result. Every threshold defaults to a no-op, so calling `filter()` with no arguments returns an equivalent list and a legitimate single-row or single-column table is never dropped unless you ask for it.

Parameters

- **min_rows** (*int, optional (default: 1)*) – Drop tables with fewer than this many rows.
- **min_columns** (*int, optional (default: 1)*) – Drop tables with fewer than this many columns.
- **min_accuracy** (*float, optional (default: 0.0)*) – Drop tables whose `parsing_report` accuracy (0-100) is below this value.
- **max_whitespace** (*float, optional (default: 100.0)*) – Drop tables whose `parsing_report` whitespace (0-100) is above this value.

Returns

A new list (the original is left untouched), so calls compose: `tables.filter(min_rows=2).filter(min_accuracy=90)`.

Return type

camelot.core.TableList

property n: int

The number of tables in the list.

stack_contiguous(*match: str = 'column_count'*, *keep_first_header: bool = False*) → *TableList*

Vertically stack tables that look like continuations across pages.

Many PDF reports break a single logical table over several pages — a header on every page, a footer on every page, body rows in between. `read_pdf` returns one *Table* per page; this helper stitches contiguous ones back together so the resulting *TableList* has one entry per *logical* table instead of one per *physical* page.

Parameters

- **match** (*str, optional (default: 'column_count')*) – How to decide whether two adjacent tables are continuations of each other.
  - `'column_count'` — same number of columns (the rule from #628's POC; the common case).
  - `'first_row'` — same column count *and* identical text in the first row (catches PDFs that repeat the header on every page).
- **keep_first_header** (*bool, optional (default: False)*) – When `match='first_row'`, the matching first row of every continuation table is dropped (so the stacked table has exactly one header row). Set to `True` to keep every page's header row in the stacked output.

Returns
A new TableList with continuation runs collapsed. Tables that don't continue from the previous one (different column count or different first row, depending on `match`) are passed through unchanged. The originals in `self` are not mutated.

Return type
camelot.core.TableList

Notes

Originally proposed by @TimothyOfDelphi in #628. Consolidates #8 / #133 / #357 / #531.

Limitations:

- The stacked *Table*'s `page` keeps the *first* stitched table's page number; `order` keeps the first table's order. Callers iterating with both shouldn't be surprised by a missing row-of-pages.
- `parsing_report` is averaged: `accuracy` and `whitespace` are mean-aggregated across the stitched tables; `confidence` is recomputed from the averaged accuracy + whitespace.
- Cell geometry (`_bbox`, `cells`, `rows`) is preserved via the y-shift trick from #628's POC, so downstream plotting on a single stitched table still works page-locally. Cells from the second-and-later tables are shifted to sit below the first table's bottom.

`class camelot.core.Table(cols, rows)`

Defines a table with coordinates relative to a left-bottom origin.

(PDF coordinate space)

Parameters
- **cols** (*list*) – List of tuples representing column x-coordinates in increasing order.
- **rows** (*list*) – List of tuples representing row y-coordinates in decreasing order.

df

Type
pandas.DataFrame

shape

Shape of the table.

Type
tuple

accuracy

Accuracy with which text was assigned to the cell.

Type
float

`whitespace`

Percentage of whitespace in the table.

Type
float

`filename`

Path of the original PDF

Type
str

`order`

Table number on PDF page.

Type
int

`page`

PDF page number.

Type
int

`property confidence: float`

A unified per-table quality score in `[0.0, 1.0]`.

Computed from the existing per-flavor signals as `(accuracy / 100) * (1 - whitespace / 100)`. The intent is a single number suitable for production filtering and automated validation — `confidence >= 0.8` works as a reasonable first-cut threshold; tune for the source PDFs.

Components and their meaning are identical across flavors:

- `accuracy` (0-100): how well the detected cells line up with the parser's structural hints (line joints for lattice, text alignments for stream/network/hybrid). Higher is better.
- `whitespace` (0-100): percentage of cells that are empty after stripping. Lower is better (a perfectly populated table is 0; a mostly-empty one trends toward 100).
- `confidence` (0-1): the composite. `accuracy=90`, `whitespace=10` → `confidence0.81`; either signal going to its worst value pulls `confidence` to 0.

See #659.

`copy_spanning_text`(*copy_text=None*)

Copies over text in empty spanning cells.

Parameters
copy_text (*list of str, optional (default: None)*) – Select one or more of the following strings: {'h', 'v'} to specify the direction in which text should be copied over when a cell spans multiple rows or columns.

Returns
The updated table with copied text in spanning cells.

Return type
camelot.core.Table

Notes

Iterates the directional copy passes until the table is stable. A single pass-per-direction misses cells spanned in *both* directions (a 2D span): the source cell from which the 2D-spanned cell would copy hasn't itself been filled yet, so the empty string propagates through. Repeating the chosen passes until no cell changes converges in O(spans) iterations and fixes the symptom reported in #349.

property data

Returns two-dimensional list of strings in table.

get_pdf_image()

Compute pdf image and cache it.

property parsing_report

Per-table parsing report.

Standard keys across all flavors:

`page`	1-based page number the table was found on.
`order`	1-based rank within that page (left-to-right / top-to-bottom).
`accuracy`	Float in `[0, 100]`. See `confidence` for component-by-component definitions.
`whitespace`	Float in `[0, 100]`.
`confidence`	Float in `[0, 1]`. Unified quality score — combines accuracy and whitespace.

See #659.

set_all_edges()

Set all table edges to True.

set_border()

Sets table border edges to True.

set_edges(*vertical*, *horizontal*, *joint_tol=2*)

Set the edges of the joint.

Set a cell's edges to True depending on whether the cell's coordinates overlap with the line's coordinates within a tolerance.

Parameters

- **vertical** (*list*) – List of detected vertical lines.
- **horizontal** (*list*) – List of detected horizontal lines.
- **joint_tol** (*int, optional*) – Tolerance for determining proximity, by default 2

to_csv(*path*, ***kwargs*)

Write Table(s) to a comma-separated values (csv) file.

For kwargs, check `pandas.DataFrame.to_csv()`.

Parameters
path (*str*) – Output filepath.

to_excel(*path*, ***kwargs*)

Write Table(s) to an Excel file.

For kwargs, check `pandas.DataFrame.to_excel()`. The optional `mode` kwarg is forwarded to `pandas.ExcelWriter` (`"w"` to overwrite, `"a"` to append a new sheet to an existing workbook) — see #317.

Parameters

- **path** (*str*) – Output filepath.
- **mode** (*str, optional (default: 'w')*) – ExcelWriter open mode. Use "a" to add a new sheet to an existing workbook (requires openpyxl).

to_html(*path*, ***kwargs*)

Write Table(s) to an HTML file.

For kwargs, check pandas.DataFrame.to_html(). The optional mode kwarg is consumed by the file-open call (#317).

Parameters

- **path** (*str*) – Output filepath.
- **mode** (*str, optional (default: 'w')*) – File open mode. Pass "a" to append to an existing file rather than overwrite it.

to_json(*path*, ***kwargs*)

Write Table(s) to a JSON file.

For kwargs, check pandas.DataFrame.to_json(). The optional mode kwarg ("w" to overwrite, "a" to append) is consumed by the file-open call; the rest are forwarded to DataFrame.to_json (#317).

Parameters

- **path** (*str*) – Output filepath.
- **mode** (*str, optional (default: 'w')*) – File open mode. Pass "a" to append to an existing file rather than overwrite it.

to_markdown(*path*, ***kwargs*)

Write Table(s) to a Markdown file.

For kwargs, check pandas.DataFrame.to_markdown(). The optional mode kwarg is consumed by the file-open call — passing mode="a" appends every successive call to the same file rather than overwriting (#317).

Parameters

- **path** (*str*) – Output filepath.
- **mode** (*str, optional (default: 'w')*) – File open mode. Pass "a" to append to an existing file rather than overwrite it.

to_sqlite(*path*, ***kwargs*)

Write Table(s) to sqlite database.

For kwargs, check pandas.DataFrame.to_sql().

Parameters

path (*str*) – Output filepath.

class camelot.core.**Cell**(*x1*, *y1*, *x2*, *y2*)

Defines a cell in a table.

With coordinates relative to a left-bottom origin. (PDF coordinate space)

Parameters

- **x1** (*float*) – x-coordinate of left-bottom point.
- **y1** (*float*) – y-coordinate of left-bottom point.
- **x2** (*float*) – x-coordinate of right-top point.

- **y2** (*float*) – y-coordinate of right-top point.

lb

Tuple representing left-bottom coordinates.

Type
tuple

lt

Tuple representing left-top coordinates.

Type
tuple

rb

Tuple representing right-bottom coordinates.

Type
tuple

rt

Tuple representing right-top coordinates.

Type
tuple

left

Whether or not cell is bounded on the left.

Type
bool

right

Whether or not cell is bounded on the right.

Type
bool

top

Whether or not cell is bounded on the top.

Type
bool

bottom

Whether or not cell is bounded on the bottom.

Type
bool

text

Text assigned to cell.

Type
string

3.1.4 Plotting

camelot.**plot**(*table*, *kind='text'*, *filename=None*, *ax=None*)

Classmethod for plotting methods.

class camelot.plotting.**PlotMethods**

Classmethod for plotting methods.

static **contour**(*table*, *ax=None*)

Generate a plot for all table boundaries present on the PDF page.

Parameters

- **table** (camelot.core.Table)
- **ax** (*matplotlib.axes.Axes (optional)*)

Returns
fig

Return type
matplotlib.fig.Figure

static **grid**(*table*, *ax=None*)

Generate a plot for the detected table grids on the PDF page.

Parameters

- **table** (camelot.core.Table)
- **ax** (*matplotlib.axes.Axes (optional)*)

Returns
fig

Return type
matplotlib.fig.Figure

static **joint**(*table*, *ax=None*)

Generate a plot for all line intersections present on the PDF page.

Parameters

- **table** (camelot.core.Table)
- **ax** (*matplotlib.axes.Axes (optional)*)

Returns
fig

Return type
matplotlib.fig.Figure

static **line**(*table*, *ax=None*)

Generate a plot for all line segments present on the PDF page.

Parameters

- **table** (camelot.core.Table)
- **ax** (*matplotlib.axes.Axes (optional)*)

Returns
fig

Return type
matplotlib.fig.Figure

static network_table_search(*table*, *ax=None*)

Generate a plot illustrating the steps of the network table search.

Parameters

- **table** (camelot.core.Table)
- **ax** (*matplotlib.axes.Axes (optional)*)

Returns
fig

Return type
matplotlib.fig.Figure

text(*table*, *ax=None*)

Generate a plot for all text elements present on the PDF page.

Parameters

- **table** (camelot.core.Table)
- **ax** (*matplotlib.axes.Axes (optional)*)

Returns
fig

Return type
matplotlib.fig.Figure

static textedge(*table*, *ax=None*)

Generate a plot for relevant textedges.

Parameters

- **table** (camelot.core.Table)
- **ax** (*matplotlib.axes.Axes (optional)*)

Returns
fig

Return type
matplotlib.fig.Figure

CHAPTER

FOUR

THE CONTRIBUTOR GUIDE

If you want to contribute to the project, this part of the documentation is for you.

4.1 Contributor's Guide

If you're reading this, you're probably looking to contributing to Camelot. *Time is the only real currency*, and the fact that you're considering spending some here is *very* generous of you. Thank you very much!

This document will help you get started with contributing documentation, code, testing and filing issues.

4.1.1 Code Of Conduct

The following quote sums up the **Code Of Conduct**.

> **Be cordial or be on your way**. *–Kenneth Reitz*

Kenneth Reitz has also written an essay on this topic, which you should read.

As the Requests Code Of Conduct states, **all contributions are welcome**, as long as everyone involved is treated with respect.

4.1.2 Your first contribution

A great way to start contributing to Camelot is to pick an issue tagged with the help wanted or the good first issue tags. If you're unable to find a good first issue, feel free to contact the maintainer.

4.1.3 Setting up a development environment

To install the dependencies needed for development, you can use pip:

```
$ pip install "camelot-py[dev]"
```

Alternatively, you can clone the project repository, and install using pip:

```
$ pip install ".[dev]"
```

4.1.4 Pull Requests

Submit a pull request

The preferred workflow for contributing to Camelot is to fork the project repository on GitHub, clone, develop on a branch and then finally submit a pull request. Here are the steps:

1. Fork the project repository. Click on the 'Fork' button near the top of the page. This creates a copy of the code under your account on the GitHub.

2. Clone your fork of Camelot from your GitHub account:

```
$ git clone https://www.github.com/[username]/camelot
```

3. Create a branch to hold your changes:

```
$ git checkout -b my-feature
```

Always branch out from `master` to work on your contribution. It's good practice to never work on the `master` branch!

> **Note**
>
> `git stash` is a great way to save the work that you haven't committed yet, to move between branches.

4. Work on your contribution. Add changed files using `git add` and then `git commit` them:

```
$ git add modified_files
$ git commit
```

5. Finally, push them to your GitHub fork:

```
$ git push -u origin my-feature
```

Now it's time to go to the your fork of Camelot and create a pull request! You can follow these instructions to do the same.

Work on your pull request

We recommend that your pull request complies with the following guidelines:

- Make sure your code follows pep8.
- In case your pull request contains function docstrings, make sure you follow the numpydoc format. All function docstrings in Camelot follow this format. Following the format will make sure that the API documentation is generated flawlessly.
- **Make sure your commit messages follow the seven rules of a great git commit message:**
  - Separate subject from body with a blank line
  - Limit the subject line to 50 characters
  - Capitalize the subject line
  - Do not end the subject line with a period
  - Use the imperative mood in the subject line
  - Wrap the body at 72 characters
  - Use the body to explain what and why vs. how
- Please prefix your title of your pull request with [MRG] (Ready for Merge), if the contribution is complete and ready for a detailed review. An incomplete pull request's title should be prefixed with [WIP] (to indicate a work in progress), and changed to [MRG] when it's complete. A good task list in the PR description will ensure that other people get a fair idea of what it proposes to do, which will also increase collaboration.
- If contributing new functionality, make sure that you add a unit test for it, while making sure that all previous tests pass. Camelot uses pytest for testing. Tests can be run using:

```
$ python setup.py test
```

4.1.5 Writing Documentation

Writing documentation, function docstrings, examples and tutorials is a great way to start contributing to open-source software! The documentation is present inside the `docs/` directory of the source code repository.

The documentation is written in reStructuredText, with Sphinx used to generate these lovely HTML files that you're currently reading (unless you're reading this on GitHub). You can edit the documentation using any text editor and then generate the HTML output by running *make html* in the `docs/` directory.

The function docstrings are written using the numpydoc extension for Sphinx. Make sure you check out how its format guidelines before you start writing one.

4.1.6 Filing Issues

We use GitHub issues to keep track of all issues and pull requests. Before opening an issue (which asks a question or reports a bug), please use GitHub search to look for existing issues (both open and closed) that may be similar.

Questions

Please don't use GitHub issues for support questions. A better place for them would be Stack Overflow. Make sure you tag them using the `python-camelot` tag.

Bug Reports

In bug reports, make sure you include:

- Your operating system type and Python version number, along with the version numbers of NumPy, OpenCV and Camelot. You can use the following code snippet to find this information:

```
import platform; print(platform.platform())
import sys; print('Python', sys.version)
import numpy; print('NumPy', numpy.__version__)
import cv2; print('OpenCV', cv2.__version__)
import camelot; print('Camelot', camelot.__version__)
```

- The complete traceback. Just adding the exception message or a part of the traceback won't help us fix your issue sooner.
- Steps to reproduce the bug, using code snippets. See Creating and highlighting code blocks.
- A link to the PDF document that you were trying to extract tables from, telling us what you expected the code to do and what actually happened.

4.2 Making a New Release

This document outlines the process for creating a new release of *camelot-py*.

The release process is semi-automated, starting with a version number change and concluding with a manual release publication on GitHub. This approach ensures versioning is correct and secure.

4.2.1 Release Steps

1. **Create a Version Bump Pull Request**

   To begin a new release, a contributor must create a pull request that increments the version number. The version number must be updated in the *pyproject.toml* file.

   For example, to release version *1.0.2*, you would change the following line in the file:

   ```
   version = "1.0.1"
   ```

   to:

   ```
   version = "1.0.2"
   ```

   The title of the pull request should be descriptive, for example: "Bump version to 1.0.2 for release".

2. **Merge the Pull Request**

   Once the pull request is reviewed and approved, it can be merged into the *main* branch. This action will trigger the *release-drafter* GitHub Action, which will automatically create a draft release with compiled notes from the merged pull requests.

3. **Publish the GitHub Release**

   This is the **manual and most critical step**. A repository maintainer must go to the *Releases* page on GitHub and find the newly created draft release.

   - Review the automatically generated release notes.
   - Ensure the tag is correct (e.g., *v1.0.2*).
   - Click the **"Publish release"** button.

4. **Automated Release Workflow**

   Publishing the release will trigger the main release workflow. This workflow will perform the following steps: - Build the Python distribution (wheel and source tarball) from the tagged commit. - Sign the distributions using Sigstore for enhanced supply chain security. - Publish the signed distributions to PyPI. - Upload the signed distributions and their signatures to the GitHub release assets.

You can monitor the progress of this workflow under the "Actions" tab of the repository.

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

J

L

M

N

O

P

R

S

T

W