sheetparser Documentation

Release 0.1a1

Guillaume Coffin

May 27, 2021

Contents

sheetparser is a library for extracting information from Excel sheets (as well as csv and pdf) that contain complex or variable layouts of tables.

Obtaining data from various sources can be very painful, and loading Excel sheets that were designed by humans for humans is especially difficult. The focus of the persons who create those sheet is first to display the information in a way that pleases their eyes or can convince others, and readability by a computer is low on the list of priorities. Also as time goes, they add intermediate lines or columns, or add more information. The systematic loading of historical information can then become a very heavy task.

The purpose of this package is to simplify the data extraction of those tables. Complex and flexible layouts can be implemented in a few lines.

CHAPTER 1

Contents:

1.1 Introduction to sheetparser

The purpose of this library is to simplify complex Excel sheets reading. The idea is to describe the layout of the sheet with spatial patterns, allowing for changes in actual position of the elements.

For instance, let's assume that we need to extract data from sheet 1 on the right. That sheet could be described as:

- One table with top and left header
- Followed by and empty line
- Another table with top and left header
- An empty line
- And 2 lines

We code the pattern as follows:

```
from sheetparser import *

pattern = Sheet('sheet', Rows,
                Table,
                Empty,
                Table,
                Empty,
                Line, Line)
```

This pattern will recognize all sheets with a similar layout. The tables could be smaller or larger. The argument *Rows* is for the layout of the sheet: it would be *Columns* if the tables were aligned horizontally.

table 1	a	b	c
1	a11	b11	c11
2	a21	b21	c21
3	a31	b31	c31
table 2	a2	b2	c2
1	a12	b12	c12
2	a22	b22	c22
3	a32	b32	c32
line1			
line2			

Fig. 1: sheet 1: Example of tables inside a sheett

The pattern is used as follows:

```
wbk = load_workbook('test_
↪table1.xlsx',with_formatting=False)
context = ListContext()
pattern.
↪match_range(wbk['Sheet2'], context)
print(context.root)
```

We first open the workbook, and create a context. This object will store the parsing result. There are 2 provided context classes: *PythonObjectResult* will contain results in a hierarchy that replicates the hierarchy of patterns, and ListResult will return them as a list of values.

The next step is to check if the pattern matches the sheet. It will raise a DoesntMatchException if not. The final step to read the result from context.root.

The result is available in context. If we print it, we obtain the following value

```
{'table': [Table table ([['a11', 'b11', 'c11'], ['a21', 'b21', 'c21'],
['a31', 'b31', 'c31']]), Table table ([['a12', 'b12', 'c12'], ['a22',
'b22', 'c22'], ['a32', 'b32', 'c32']])], 'line': [['line1'],
['line2']], '__meta': [{'name': 'Sheet2'}]}
```

This represents the result of the parsing, and can be easily used in a program, for instance to load in a database.

Now let'ss assume that the accounting department issues one month later an updated version of this document (version 2), but there is now only 1 table and 3 lines. We can add a little more flexibility in our pattern:

```
pattern = Sheet('sheet', Rows,

↪    Many(Table, Empty, min=1,max=2),
            Many(Line)
            )
```

With that pattern, we can match both sheets. Alternatively, we could code this:

```
pattern = Sheet('sheet', Rows,
            Table, Empty,
            Maybe(Table, Empty),
            Many(Line)
            )
```

The idea is similar to regular expressions. It provides a powerful language to accommodate different situations, and with a flexible system to detect sheet features such as text or formatting.

table 1	a	b	c
1	a11	b11	c11
2	a21	b21	c21
3	a31	b31	c31
line1			
line2			
line3			

Fig. 2: sheet 2: Version 2 of the sheet

1.1.1 Another example

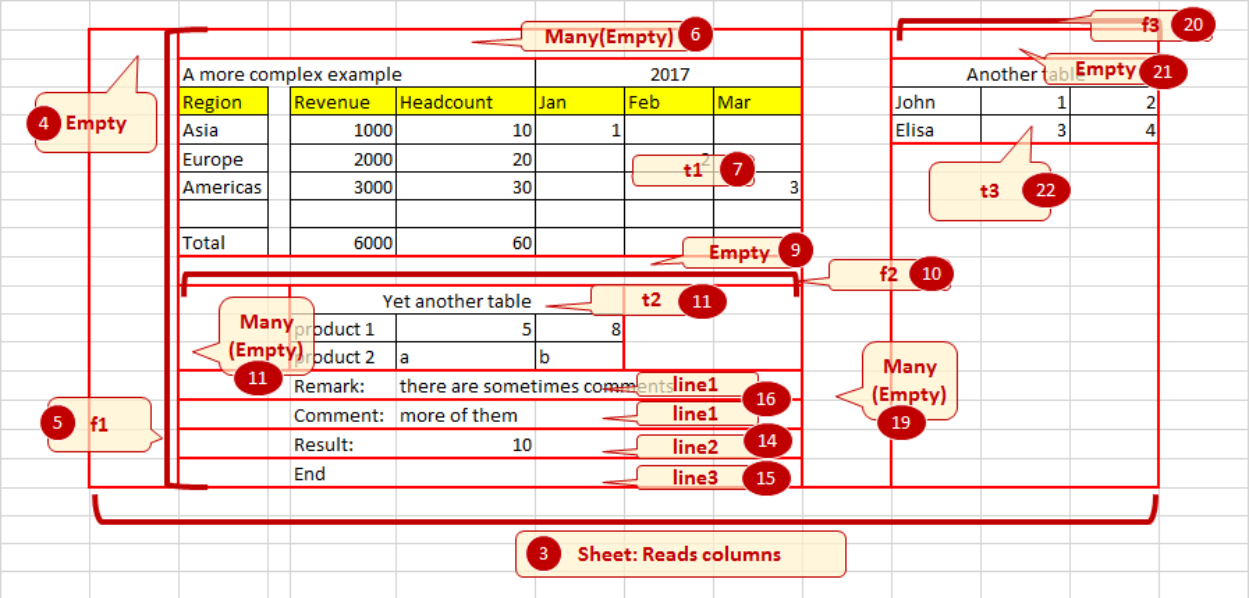
Here's a more complex example:

```
1  wbk = load_workbook(filename, with_formatting=True)
2  sheet = wbk['Sheet6']
3  pattern = Sheet('sheet', Columns,
4                  Many(Empty),
5        FlexibleRange('f1',Rows,
6          Many(Empty),
7                  Table('t1',[GetValue,HeaderTableTransform(2,1),FillData,RemoveEmptyLines('columns')],
8                  stop=no_horizontal),
9    Empty,
10         FlexibleRange('f2',Columns,
11           Many(Empty),Table('t2'),
12                 stop=no_horizontal),
13         Many(Empty),
14                 Many((Line('line2',[get_value,Match('Result:')])
15           + Line('line3',[StripLine(),get_value]))
16                 | Line('line1')),
17     stop = lambda line,linecount:linecount>2 and empty_line(line)
18                                 ),
19                 Many(Empty),
20       FlexibleRange('f3',Rows,
21         Many(Empty),
22 Table('t3',stop = no_horizontal)))
```

Here's how it works (the numers refer to the line number in the code above):

1.1.2 PDF Files

Version 0.2 includes a PDF backend. The document is seen as a Book and the Sheet are the pages of the document. Use the page number instead of the sheet name.



1.2 Patterns

1.2.1 Some definitions

range is an Excel range, delimited with a top, a left, a right and a bottom. A sheet is an example of a range.

line is a row or a column. This is decided by the chosen **layout**: horizontal layouts will yield rows, vertical will yield columns.

pattern is an object that matches the given range or line(s). If the match fails, the method raises a DoesntMatchException. If it succeeds, it fills up the context given as a parameter.

Note that patterns can be passed as arguments to the upper level pattern as object or classes. Classes will be instatianted.

There are 3 types of patterns:

1.2.2 Workbook

This pattern will be called to match a workbook:

1.2.3 Ranges

The following patterns match either the whole sheet or a range:

Layout is Rows or Columns, and will be used to know if the range should be read horizontally or vertically.

1.2.4 Iterators of lines

These patterns are called on an iterator of lines, and will be passed as parameters to Range patterns or other patterns matching iterators of lines.

These patterns can be combined with the operator +, which returns a Sequence. a+b is equivalent to Sequence(a,b). Similarly, a|b is equivalent to OrPattern(a,b).

The name of the pattern is used by the ResultContext to store the matched element. The existing patterns that operate on an line iterator are:

1.2.5 Stop tests

Stop tests are functions that are passed to FlexibleRange and Table to detect the end of a block. You can create your own and pass it as a parameter to the pattern.

1.3 Line and table transformations

The contents that is matched by the *Line* and *Table* patterns is stored in the context result. Another level of processing is provided by list of transformations.

1.3.1 Line transformations

They are passed as *line_args* parameters to the Line pattern. It is a list of function that take a list and return a list. These functions are called in sequence, the result of one function is passed to the following one.

The first function of the list must accept a list of Cell. The function get_value transforms it to the list of values.

These are the included line transformations:

Parameterized functions (objects with a method __call__):

1.3.2 Table transformations

Similarly, the lines matched by the *Table* pattern are passed to a series of processings. They are subclasses of *TableTransform* which implement *wrap* or *process_line* (or both). *process_line* is called when a new line is added, and *wrap* is called at the end when all lines have been added.

1.4 Results

When a pattern is matched, it fills a *ResultContext*. The *ResultContext* has to be instantiated by the client and passed to the match method. Here are the provided classes that derive from *ResultContext*:

1.5 Documents and backends

We load first a backend - that's the module that will read the Excel sheet and provide the information to the library. There are 5 provided backends:

- one based on xlrd, that can read xls file and xlsx without

formatting,

- one based on openpyxl that can read xlsx files with some

formatting information

- one is based on win32com and the actual Excel program, with serious performance issues
- raw provides an interface for data stored as list, as well as csv files
- pdfminer provides an interface for pdf files. This feature is experimental and is limited by the amount of information that pdf files can provide.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search