pyexcel Documentation

Release 0.2.5

Onni Software Ltd.

September 01, 2016

Contents

1 Introduction 3

2 Getting the source 5

3 Installation 7

4 Usage 9

5 Design 11
- 5.1 Introduction . . . 11
- 5.2 Signature functions . . . 13

6 Tutorial 17
- 6.1 Sheet: Data Access . . . 17
- 6.2 Sheet: Data manipulation . . . 23
- 6.3 Sheet: Data filtering . . . 26
- 6.4 Sheet: Data conversion . . . 29
- 6.5 Sheet: Formatting . . . 34
- 6.6 Book: Sheet operations . . . 36
- 6.7 Work with excel files . . . 37
- 6.8 Work with excel files in memory . . . 40
- 6.9 Migrate from 0.2.1 to 0.2.2 . . . 41
- 6.10 Migrate from 0.1.x to 0.2.x . . . 42

7 Cook book 45
- 7.1 Recipies . . . 45
- 7.2 Loading from other sources . . . 49

8 Real world cases 51
- 8.1 Questions and Answers . . . 51

9 API documentation 53
- 9.1 API Reference . . . 53
- 9.2 Internal API reference . . . 93

10 Developer's guide 103
- 10.1 Developer's guide . . . 103

11 Change log 107

11.1 Change log 107

12 Indices and tables **109**

Author C.W.

Source code http://github.com/pyexcel/pyexcel

Issues http://github.com/pyexcel/pyexcel/issues

License New BSD License

Version 0.2.5

Generated September 01, 2016

CHAPTER 1

Introduction

pyexcel provides **one** application programming interface to read, manipulate and write data in different excel formats. This library makes information processing involving excel files an enjoyable task. The data in excel files can be turned into *array or dict* with least code, vice versa. And ready-made custom *filters* and *formatters* can be applied. This library focuses on data processing using excel files as storage media hence fonts, colors and charts were not and will not be considered.

Excel files are de-facto file format for information sharing in non-software centric organisations. Excel files are not only used for mathematical computation in financial institutions but also used for many other purposes in an office work environment. This is largely casued by wide adoption of Microsoft Office. Comparing the existing, mathematics savvy Pandas library, this library intends to help data processing job where data extraction is more important than data analysis. In such context, ease of use, and low overhead is preferred, while Pandas is as big as 4MB and contains hundreds of potentially useful functions.

Note: Since version *0.2.2*, no longer a plugin should be explicitly imported. They are imported if they are installed. Please use pip to manage the plugins.

CHAPTER 2

Getting the source

Source code is hosted in github. You can get it using git client:

```
$ git clone http://github.com/pyexcel/pyexcel.git
```

Installation

You can install it via pip:

```
$ pip install pyexcel
```

or clone it and install it:

```
$ git clone http://github.com/pyexcel/pyexcel.git
$ cd pyexcel
$ python setup.py install
```

For individual excel file formats, please install them as you wish:

Table 3.1: a map of plugins and supported excel file formats

Plugin	Supported file formats	Dependencies	Python versions	Comments
pyexcel	csv, csvz [1], tsv, tsvz [2]	pyexcel-io	2.6, 2.7, 3.3, 3.4, 3.5, pypy	
xls	xls, xlsx(read only), xlsm(read only)	xlrd, xlwt	2.6, 2.7, 3.3, 3.4, 3.5, pypy	supports reading xlsx as well
xlsx	xlsx	openpyxl	2.6, 2.7, 3.3, 3.4, 3.5, pypy	
ods3	ods	ezodf, lxml	2.6, 2.7, 3.3, 3.4, 3.5,	
ods	ods (python 2.6, 2.7)	odfpy	2.6, 2.7	
text	json, rst, mediawiki, latex, grid, etc.	tabulate	2.6, 2.7, 3.3, 3.4, 3.5, pypy	writing to files only

Table 3.2: Plugin compatibility table

pyexcel	pyexcel-io	xls	xlsx	ods	ods3	text
0.2.2+	0.2.0	0.2.0	0.2.0	0.2.0	0.2.0	0.2.1+
0.2.1	0.1.0	0.1.0	0.1.0	0.1.0+	0.1.0+	0.2.0
0.2.0	0.1.0	0.1.0	0.1.0	0.1.0+	0.1.0+	0.1.0+

[1] zipped csv file
[2] zipped tsv file

CHAPTER 4

Usage

Suppose you want to process the following excel data :

Name	Age
Adam	28
Beatrice	29
Ceri	30
Dean	26

Here are the example usages:

```
>>> import pyexcel as pe
>>> records = pe.get_records(file_name="your_file.xls")
>>> for record in records:
...     print("%s is aged at %d" % (record['Name'], record['Age']))
Adam is aged at 28
Beatrice is aged at 29
Ceri is aged at 30
Dean is aged at 26
```

CHAPTER 5

Design

5.1 Introduction

This section introduces Excel data models, its representing data structures and provides an overview of formatting, transformation, manipulation supported by **pyexcel**

5.1.1 Data models and data structures

When dealing with excel files, there are three primary objects: **cell**, **sheet** and **book**. A book contains one or more sheets and a sheet is consisted of a sheet name and a two dimensional array of cells. Although a sheet can contain charts and a cell can have formular, styling properties, this library ignores them and pay attention to the data in the cell and its data type. So, in the context of this library, the definition of those three concepts are:

concept	definition	pyexcel data model
a cell	is a data unit	a Python data type
a sheet	is a named two dimensional array of data units	`Sheet`
a book	is a dictionary of two dimensional array of data units.	`Book`

5.1.2 Data source

The most popular data source is an excel file. Libre Offcie/Microsoft Excel could easily generate an new excel file of desired format. Besides a physical file, this library recognizes additional three additional sources:

1. Excel files in computer memory. For example when a file was uploaded to a Python server for information processing, if it is relatively small, it will be stored in memory.
2. Database tables. For example, a client would like to have a snapshot of some database table in an excel file and ask it to be sent to him.
3. Python structures. For example, a developer may have scrapped a site and hence stored data in Python array or dictionary. He may want to save those information as a file.

5.1.3 Data format

This library and its plugins support most of the frequently used excel file formats.

file format	defintion	Single Sheet
csv	comma separated values	Yes
tsv	tab separated values	Yes
csvz	a zip file that contains one or many csv files	
tsvz	a zip file that contains one or many tsv files	
xls	a spreadsheet file format created by MS-Excel 97-2003 [1]	
xlsx	MS-Excel Extensions to the Office Open XML SpreadsheetML File Format. [2]	
xlsm	an MS-Excel Macro-Enabled Workbook file	
ods	open document spreadsheet	
json	java script object notation	

See also *a map of plugins and supported excel file formats*.

5.1.4 Data transformation

Quite often, a developer would like to have the excel data in a Python data structures. This library supports the *conversions from* previous three data source to the following list of data strcutures, and *vice versa*.

Table 5.1: A list of supported data structures

Psudo name	Python name	Related model
two dimensional array	a list of lists	`Sheet`
a dictionary of one dimensional arrays	a dictionary of lists	`Sheet`
a list of dictionaries	a list of dictionaries	`Sheet`
a dictionary of two dimensional arrays	a dictionary of lists of lists	`Book`

Examples:

```
>>> two_dimensional_list = [
...    [1, 2, 3, 4],
...    [5, 6, 7, 8],
...    [9, 10, 11, 12],
... ]
>>> a_dictionary_of_one_dimensional_arrays = {
...     "Column 1": [1, 2, 3, 4],
...     "Column 2": [5, 6, 7, 8],
...     "Column 3": [9, 10, 11, 12],
... }
>>> a_list_of_dictionaries = [
...     {
...         "Name": 'Adam',
...         "Age": 28
...     },
...     {
...         "Name": 'Beatrice',
...         "Age": 29
...     },
...     {
...         "Name": 'Ceri',
...         "Age": 30
...     },
...     {
...         "Name": 'Dean',
...         "Age": 26
```

[1] quoted from whatis.com. Technical details can be found at MSDN XLS

[2] xlsx is used by MS-Excel 2007, more infomation can be found at MSDN XLSX

```
...     }
... ]
>>> a_dictionary_of_two_dimensional_arrays = {
...      'Sheet 1':
...          [
...              [1.0, 2.0, 3.0],
...              [4.0, 5.0, 6.0],
...              [7.0, 8.0, 9.0]
...          ],
...      'Sheet 2':
...          [
...              ['X', 'Y', 'Z'],
...              [1.0, 2.0, 3.0],
...              [4.0, 5.0, 6.0]
...          ],
...      'Sheet 3':
...          [
...              ['O', 'P', 'Q'],
...              [3.0, 2.0, 1.0],
...              [4.0, 3.0, 2.0]
...          ]
...  }
```

5.1.5 Data manipulations

The main operation on a cell involves *cell access*, *formatting* and *cleansing*. The main operation on a sheet involves the group access to a row or a column, data filtering and data transformation. The main operation in a book is obtain access to individual sheets.

5.2 Signature functions

5.2.1 Import data into Python

This library provides one application programming interface to read data from one of the following data sources:

- physical file
- memory file
- SQLAlchemy table
- Django Model
- Python data stuctures: dictionary, records and array

and to transform them into one of the data structures:

- two dimensinal array
- a dictionary of one dimensional arrays
- a list of dictionaries
- a dictionary of two dimensional arrays
- a *Sheet*
- a *Book*

Four data access functions

It is believed that once a Python developer could easily operate on list, dictionary and various mixture of both. This library provides four module level functions to help you obtain excel data in those formats. Please refer to "A list of module level functions", the first three functions operates on any one sheet from an excel book and the fourth one returns all data in all sheets in an excel book.

Table 5.2: A list of module level functions

Functions	Psudo name	Python name
`get_array()`	two dimensional array	a list of lists
`get_dict()`	a dictionary of one dimensional arrays	an ordered dictionary of lists
`get_records()`	a list of dictionaries	a list of dictionaries
`get_book_dict()`	a dictionary of two dimensional arrays	a dictionary of lists of lists

See also:

- *How to get an array from an excel sheet*
- *How to get a dictionary from an excel sheet*
- *How to obtain records from an excel sheet*
- *How to obtain a dictionary from a multiple sheet book*

Two native functions

In cases where the excel data needs custom manipulations, a pyexcel user got a few choices: one is to use `Sheet` and `Book`, the other is to look for more sophisticated ones:

- Pandas, for numerical analysis
- Do-it-yourself

Functions	Returns
`get_sheet()`	`Sheet`
`get_book()`	`Book`

For all six functions, you can pass on the same command parameters while the return value is what the function says.

5.2.2 Export data from Python

This library provides one application programming interface to transform them into one of the data structures:

- two dimensinal array
- a (ordered) dictionary of one dimensional arrays
- a list of dictionaries
- a dictionary of two dimensional arrays
- a `Sheet`
- a `Book`

and write to one of the following data sources:

- physical file
- memory file

- SQLAlchemy table
- Django Model
- Python data stuctures: dictionary, records and array

Here are the two functions:

Functions	Description
`save_as()`	Works well with single sheet file
`save_book_as()`	Works with multiple sheet file

See also:

- *How to save an python array as an excel file*
- *How to save a dictionary of two dimensional array as an excel file*
- *How to save an python array as a csv file with special delimiter*

5.2.3 Data transportation/transcoding

Based the capability of this library, it is capable of transporting your data in between any of these data sources:

- physical file
- memory file
- SQLAlchemy table
- Django Model
- Python data stuctures: dictionary, records and array

See also:

- *How to an excel sheet to a database using SQLAlchemy*
- *How to open an xls file and save it as xlsx*
- *How to open an xls file and save it as csv*

CHAPTER 6

Tutorial

6.1 Sheet: Data Access

6.1.1 Random access to individual cell

To randomly access a cell of *Sheet* instance, two syntax are available:

```
sheet[row, column]
```

or:

```
sheet['A1']
```

The former syntax is handy when you know the row and column numbers. The latter syntax is introduced to help you convert the excel column header such as "AX" to integer numbers.

Suppose you have the following data, you can get value 5 by reader[2, 2].

Example	X	Y	Z
a	1	2	3
b	4	5	6
c	7	8	9

Here is the example code showing how you can randomly access a cell:

```
>>> import pyexcel
```

```
>>> sheet = pyexcel.get_sheet(file_name="example.xls")
>>> sheet.content
+---------+---+---+---+
| Example | X | Y | Z |
+---------+---+---+---+
| a       | 1 | 2 | 3 |
+---------+---+---+---+
| b       | 4 | 5 | 6 |
+---------+---+---+---+
| c       | 7 | 8 | 9 |
+---------+---+---+---+
>>> print(sheet[2, 2])
5
>>> print(sheet["C3"])
5
```

6.1.2 Random access to rows and columns

Continue with previous excel file, you can access row and column separately:

```
>>> sheet.row[1]
['a', 1, 2, 3]
>>> sheet.column[2]
['Y', 2, 5, 8]
```

6.1.3 Use custom names instead of index

Alternatively, it is possible to use the first row to refer to each columns:

```
>>> sheet.name_columns_by_row(0)
>>> print(sheet[1, "Y"])
5
```

You have noticed the row index has been changed. It is because first row is taken as the column names, hence all rows after the first row are shifted. Now accessing the columns are changed too:

```
>>> sheet.column['Y']
[2, 5, 8]
```

Hence access the same cell, this statement also works:

```
>>> sheet.column['Y'][1]
5
```

Further more, it is possible to use first column to refer to each rows:

```
>>> sheet.name_rows_by_column(0)
```

To access the same cell, we can use this line:

```
>>> sheet.row["b"][1]
5
```

For the same reason, the row index has been reduced by 1. Since we have named columns and rows, it is possible to access the same cell like this:

```
>>> print(sheet["b", "Y"])
5
```

For multiple sheet file, you can regard it as three dimensional array if you use `Book`. So, you access each cell via this syntax:

```
book[sheet_index][row, column]
```

or:

```
book["sheet_name"][row, column]
```

Suppose you have the following sheets:

Table 6.1: Sheet 1

1	2	3
4	5	6
7	8	9

Table 6.2: Sheet 2

X	Y	Z
1	2	3
4	5	6

Table 6.3: Sheet 3

O	P	Q
3	2	1
4	3	2

And you can randomly access a cell in a sheet:

```
>>> book = pyexcel.get_book(file_name="example.xls")
>>> print(book["Sheet 1"][0,0])
1
>>> print(book[0][0,0]) # the same cell
1
```

Tip: With pyexcel, you can regard single sheet reader as an two dimensional array and multi-sheet excel book reader as a ordered dictionary of two dimensional arrays.

6.1.4 Reading a single sheet excel file

Suppose you have a csv, xls, xlsx file as the following:

1	2	3
4	5	6
7	8	9

The following code will give you the data in json:

```
>>> import json
>>> # "example.csv","example.xlsx","example.xlsm"
>>> sheet = pyexcel.get_sheet(file_name="example.xls")
>>> print(json.dumps(sheet.to_array()))
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Read the sheet as a dictionary

Suppose you have a csv, xls, xlsx file as the following:

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

The following code will give you data series in a dictionary:

```
>>> # "example.xls","example.xlsx","example.xlsm"
>>> sheet = pyexcel.get_sheet(file_name="example_series.xls", name_columns_by_row=0)
```

```
>>> sheet.to_dict()
OrderedDict([('Column 1', [1, 4, 7]), ('Column 2', [2, 5, 8]), ('Column 3', [3, 6, 9])])
```

Can I get an array of dictionaries per each row?

Suppose you have the following data:

X	Y	Z
1	2	3
4	5	6
7	8	9

The following code will produce what you want:

```
>>> # "example.csv","example.xlsx","example.xlsm"
>>> sheet = pyexcel.get_sheet(file_name="example.xls", name_columns_by_row=0)
>>> records = sheet.to_records()
>>> for record in records:
...     keys = sorted(record.keys())
...     print("{")
...     for key in keys:
...         print("'%s':%d" % (key, record[key]))
...     print("}")
{
'X':1
'Y':2
'Z':3
}
{
'X':4
'Y':5
'Z':6
}
{
'X':7
'Y':8
'Z':9
}
>>> print(records[0]["X"]) # access first row and first item
1
```

6.1.5 Writing a single sheet excel file

Suppose you have an array as the following:

1	2	3
4	5	6
7	8	9

The following code will write it as an excel file of your choice:

```
.. testcode::
```

```
>>> array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> # "output.xls" "output.xlsx" "output.ods" "output.xlsm"
>>> sheet = pyexcel.Sheet(array)
>>> sheet.save_as("output.csv")
```

Suppose you have a dictionary as the following:

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

The following code will write it as an excel file of your choice:

```
>>> example_dict = {"Column 1": [1, 2, 3], "Column 2": [4, 5, 6], "Column 3": [7, 8, 9]}
>>> # "output.xls" "output.xlsx" "output.ods" "output.xlsm"
>>> sheet = pyexcel.get_sheet(adict=example_dict)
>>> sheet.save_as("output.csv")
```

6.1.6 Write multiple sheet excel file

Suppose you have previous data as a dictionary and you want to save it as multiple sheet excel file:

```
>>> content = {
...     'Sheet 1':
...         [
...             [1.0, 2.0, 3.0],
...             [4.0, 5.0, 6.0],
...             [7.0, 8.0, 9.0]
...         ],
...     'Sheet 2':
...         [
...             ['X', 'Y', 'Z'],
...             [1.0, 2.0, 3.0],
...             [4.0, 5.0, 6.0]
...         ],
...     'Sheet 3':
...         [
...             ['O', 'P', 'Q'],
...             [3.0, 2.0, 1.0],
...             [4.0, 3.0, 2.0]
...         ]
... }
>>> book = pyexcel.get_book(bookdict=content)
>>> book.save_as("output.xls")
```

You shall get a xls file

6.1.7 Read multiple sheet excel file

Let’s read the previous file back:

```
>>> book = pyexcel.get_book(file_name="output.xls")
>>> sheets = book.to_dict()
>>> for name in sheets.keys():
...     print(name)
Sheet 1
Sheet 2
Sheet 3
```

6.1.8 Work with data series in a single sheet

Suppose you have the following data in any of the supported excel formats again:

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

```
>>> sheet = pyexcel.get_sheet(file_name="example_series.xls", name_columns_by_row=0)
```

Play with data

You can get headers:

```
>>> print(list(sheet.colnames))
['Column 1', 'Column 2', 'Column 3']
```

You can use a utility function to get all in a dictionary:

```
>>> sheet.to_dict()
OrderedDict([('Column 1', [1, 4, 7]), ('Column 2', [2, 5, 8]), ('Column 3', [3, 6, 9])])
```

Maybe you want to get only the data without the column headers. You can call `rows()` instead:

```
>>> pyexcel.to_array(sheet.rows())
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

You can get data from the bottom to the top one by calling `rrows()` instead:

```
>>> pyexcel.to_array(sheet.rrows())
[[7, 8, 9], [4, 5, 6], [1, 2, 3]]
```

You might want the data arranged vertically. You can call `columns()` instead:

```
>>> pyexcel.to_array(sheet.columns())
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

You can get columns in reverse sequence as well by calling `rcolumns()` instead:

```
>>> pyexcel.to_array(sheet.rcolumns())
[[3, 6, 9], [2, 5, 8], [1, 4, 7]]
```

Do you want to flatten the data? You can get the content in one dimensional array. If you are interested in playing with one dimensional enumeration, you can check out these functions `enumerate()`, `reverse()`, `vertical()`, and `rvertical()`:

```
>>> pyexcel.to_array(sheet.enumerate())
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> pyexcel.to_array(sheet.reverse())
[9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> pyexcel.to_array(sheet.vertical())
[1, 4, 7, 2, 5, 8, 3, 6, 9]
>>> pyexcel.to_array(sheet.rvertical())
[9, 6, 3, 8, 5, 2, 7, 4, 1]
```

6.2 Sheet: Data manipulation

The data in a sheet is represented by *Sheet* which maintains the data as a list of lists. You can regard *Sheet* as a two dimensional array with additional iterators. Random access to individual column and row is explosed by *NamedColumn* and *NamedRow*

6.2.1 Column manipulation

Suppose have one data file as the following:

```
>>> sheet = pyexcel.get_sheet(file_name="example.xls", name_columns_by_row=0)
>>> sheet
pyexcel sheet:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+==========+==========+==========+
| 1        | 4        | 7        |
+----------+----------+----------+
| 2        | 5        | 8        |
+----------+----------+----------+
| 3        | 6        | 9        |
+----------+----------+----------+
```

And you want to update `Column 2` with these data: [11, 12, 13]

```
>>> sheet.column["Column 2"] = [11, 12, 13]
>>> sheet.column[1]
[11, 12, 13]
>>> sheet
pyexcel sheet:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+==========+==========+==========+
| 1        | 11       | 7        |
+----------+----------+----------+
| 2        | 12       | 8        |
+----------+----------+----------+
| 3        | 13       | 9        |
+----------+----------+----------+
```

Remove one column of a data file

If you want to remove `Column 2`, you can just call:

```
>>> del sheet.column["Column 2"]
>>> sheet.column["Column 3"]
[7, 8, 9]
```

The sheet content will become:

```
>>> sheet
pyexcel sheet:
+----------+----------+
| Column 1 | Column 3 |
+==========+==========+
| 1        | 7        |
```

```
+----------+----------+
| 2        | 8        |
+----------+----------+
| 3        | 9        |
+----------+----------+
```

6.2.2 Append more columns to a data file

Continue from previous example. Suppose you want add two more columns to the data file

Column 4	Column 5
10	13
11	14
12	15

Here is the example code to append two extra columns:

```
>>> extra_data = [
...     ["Column 4", "Column 5"],
...     [10, 13],
...     [11, 14],
...     [12, 15]
... ]
>>> sheet2 = pyexcel.Sheet(extra_data)
>>> sheet.column += sheet2
>>> sheet.column["Column 4"]
[10, 11, 12]
>>> sheet.column["Column 5"]
[13, 14, 15]
```

Here is what you will get:

```
>>> sheet
pyexcel sheet:
+----------+----------+----------+----------+
| Column 1 | Column 3 | Column 4 | Column 5 |
+==========+==========+==========+==========+
| 1        | 7        | 10       | 13       |
+----------+----------+----------+----------+
| 2        | 8        | 11       | 14       |
+----------+----------+----------+----------+
| 3        | 9        | 12       | 15       |
+----------+----------+----------+----------+
```

Cherry pick some columns to be removed

Suppose you have the following data:

```
>>> data = [
...      ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'],
...      [1,2,3,4,5,6,7,9],
... ]
>>> sheet = pyexcel.Sheet(data, name_columns_by_row=0)
>>> sheet
pyexcel sheet:
+---+---+---+---+---+---+---+---+
| a | b | c | d | e | f | g | h |
```

```
+===+===+===+===+===+===+===+===+
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
+---+---+---+---+---+---+---+---+
```

And you want to remove columns named as: 'a', 'c, 'e', 'h'. This is how you do it:

```
>>> del sheet.column['a', 'c', 'e', 'h']
>>> sheet
pyexcel sheet:
+---+---+---+---+
| b | d | f | g |
+===+===+===+===+
| 2 | 4 | 6 | 7 |
+---+---+---+---+
```

6.2.3 What if the headers are in a different row

Suppose you have the following data:

```
>>> sheet
pyexcel sheet:
+----------+----------+----------+
| 1        | 2        | 3        |
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+----------+----------+----------+
| 4        | 5        | 6        |
+----------+----------+----------+
```

The way to name your columns is to use index 1:

```
>>> sheet.name_columns_by_row(1)
```

Here is what you get:

```
>>> sheet
pyexcel sheet:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+==========+==========+==========+
| 1        | 2        | 3        |
+----------+----------+----------+
| 4        | 5        | 6        |
+----------+----------+----------+
```

6.2.4 Row manipulation

Suppose you have the following data:

```
>>> sheet
pyexcel sheet:
+---+---+---+-------+
| a | b | c | Row 1 |
+---+---+---+-------+
| e | f | g | Row 2 |
+---+---+---+-------+
```

```
| 1 | 2 | 3 | Row 3 |
+---+---+---+-------+
```

You can name your rows by column index at 3:

```
>>> sheet.name_rows_by_column(3)
```

Then you can access rows by its name:

```
>>> sheet.row["Row 1"]
['a', 'b', 'c']
```

6.3 Sheet: Data filtering

There are two ways of applying a filter:

1. soft filtering. use *add_filter()*, *remove_filter()* and *clear_filters()* to interactively apply a filter. The content is not modified until you call *freeze_filters()*
2. hard filtering. use *filter()* function to apply a filter immediately. The content is modified.

Suppose you have the following data in any of the supported excel formats:

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

```
>>> import pyexcel
```

```
>>> sheet = pyexcel.get_sheet(file_name="example_series.xls", name_columns_by_row=0)
```

6.3.1 Filter out some data

You may want to filter odd rows and print them in an array of dictionaries:

```
>>> sheet.add_filter(pyexcel.OddRowFilter())
>>> sheet.to_array()
[['Column 1', 'Column 2', 'Column 3'], [4, 5, 6]]
```

Let's try to further filter out even columns:

```
>>> sheet.add_filter(pyexcel.EvenColumnFilter())
>>> sheet.to_dict()
OrderedDict([('Column 1', [4]), ('Column 3', [6])])
```

Save the data

Let's save the previous filtered data:

```
>>> sheet.save_as("example_series_filter.xls")
```

When you open *example_series_filter.xls*, you will find these data

Column 1	Column 3
2	8

The complete code is:

```
import pyexcel

sheet = pyexcel.get_sheet(file_name="example_series.xls")
sheet.add_filter(pyexcel.OddRowFilter())
sheet.add_filter(pyexcel.EvenColumnFilter())
sheet.save_as("example_series_filter.xls")
```

How to filter out empty rows in my sheet?

Suppose you have the following data in a sheet and you want to remove those rows with blanks:

```
>>> import pyexcel as pe
>>> sheet = pe.Sheet([[1,2,3],['','',''],['','',''],[1,2,3]])
>>> sheet
pyexcel sheet:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
+---+---+---+
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
```

You can use *pyexcel.filters.RowValueFilter*, which examines each row, return *True* if the row should be filtered out. So, let's define a filter function:

```
>>> def filter_row(row):
...     result = [element for element in row if element != '']
...     return len(result)==0
```

Now, let's contruct a row value filter

```
>>> row_value_filter = pe.RowValueFilter(filter_row)
```

And then apply the filter on the sheet:

```
>>> sheet.filter(row_value_filter)
>>> sheet
pyexcel sheet:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
```

6.3.2 Work with multi-sheet file

How do I read a book, pocess it and save to a new book

Yes, you can do that. The code looks like this:

```
import pyexcel

book = pyexcel.get_book(file_name="yourfile.xls")
for sheet in book:
```

```
    # do you processing with sheet
    # do filtering?
    pass
book.save_as("output.xls")
```

What would happen if I save a multi sheet book into "csv" file

Well, you will get one csv file per each sheet. Suppose you have these code:

```
>>> content = {
...     'Sheet 1':
...         [
...             [1.0, 2.0, 3.0],
...             [4.0, 5.0, 6.0],
...             [7.0, 8.0, 9.0]
...         ],
...     'Sheet 2':
...         [
...             ['X', 'Y', 'Z'],
...             [1.0, 2.0, 3.0],
...             [4.0, 5.0, 6.0]
...         ],
...     'Sheet 3':
...         [
...             ['O', 'P', 'Q'],
...             [3.0, 2.0, 1.0],
...             [4.0, 3.0, 2.0]
...         ]
... }
>>> book = pyexcel.Book(content)
>>> book.save_as("myfile.csv")
```

You will end up with three csv files:

```
>>> import glob
>>> outputfiles = glob.glob("myfile_*.csv")
>>> for file in sorted(outputfiles):
...     print(file)
...
myfile__Sheet 1__0.csv
myfile__Sheet 2__1.csv
myfile__Sheet 3__2.csv
```

and their content is the value of the dictionary at the corresponding key

After I have saved my multiple sheet book in csv format, how do I get them back in pyexcel

First of all, you can read them back individual as csv file using *meth:~pyexcel.get_sheet* method. Secondly, the pyexcel can do the magic to load all of them back into a book. You will just need to provide the common name before the separator "__":

```
>>> book2 = pyexcel.get_book(file_name="myfile.csv")
>>> book2
Sheet 1:
+---+---+---+
| 1 | 2 | 3 |
```

```
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 7 | 8 | 9 |
+---+---+---+
Sheet 2:
+---+---+---+
| X | Y | Z |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
Sheet 3:
+---+---+---+
| O | P | Q |
+---+---+---+
| 3 | 2 | 1 |
+---+---+---+
| 4 | 3 | 2 |
+---+---+---+
```

6.4 Sheet: Data conversion

6.4.1 How to obtain records from an excel sheet

Suppose you want to process the following excel data :

Name	Age
Adam	28
Beatrice	29
Ceri	30
Dean	26

Here are the example code:

```
>>> import pyexcel as pe
>>> records = pe.get_records(file_name="your_file.xls")
>>> for record in records:
...     print("%s is aged at %d" % (record['Name'], record['Age']))
Adam is aged at 28
Beatrice is aged at 29
Ceri is aged at 30
Dean is aged at 26
```

6.4.2 How to get an array from an excel sheet

Suppose you have a csv, xls, xlsx file as the following:

1	2	3
4	5	6
7	8	9

The following code will give you the data in json:

```
>>> import pyexcel
>>> # "example.csv","example.xlsx","example.xlsm"
>>> my_array = pyexcel.get_array(file_name="example.xls")
>>> my_array
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

6.4.3 How to save an python array as an excel file

Suppose you have the following array:

```
>>> data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

And here is the code to save it as an excel file

```
>>> import pyexcel
>>> pyexcel.save_as(array=data, dest_file_name="example.xls")
```

Let's verify it:

```
>>> pyexcel.get_sheet(file_name="example.xls")
pyexcel_sheet1:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 7 | 8 | 9 |
+---+---+---+
```

6.4.4 How to save an python array as a csv file with special delimiter

Suppose you have the following array:

```
>>> data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

And here is the code to save it as an excel file

```
>>> import pyexcel
>>> pyexcel.save_as(array=data,
...                 dest_file_name="example.csv",
...                 dest_delimiter=':')
```

Let's verify it:

```
>>> with open("example.csv") as f:
...     for line in f.readlines():
...         print(line.rstrip())
...
1:2:3
4:5:6
7:8:9
```

6.4.5 How to get a dictionary from an excel sheet

Suppose you have a csv, xls, xlsx file as the following:

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

The following code will give you data series in a dictionary:

```
>>> import pyexcel
>>> from pyexcel._compact import OrderedDict
>>> my_dict = pyexcel.get_dict(file_name="example_series.xls", name_columns_by_row=0)
>>> isinstance(my_dict, OrderedDict)
True
>>> for key, values in my_dict.items():
...     print({key: values})
{'Column 1': [1, 4, 7]}
{'Column 2': [2, 5, 8]}
{'Column 3': [3, 6, 9]}
```

Please note that my_dict is an OrderedDict.

6.4.6 How to obtain a dictionary from a multiple sheet book

Suppose you have a multiple sheet book as the following:

Table 6.4: Sheet 1

1	2	3
4	5	6
7	8	9

Table 6.5: Sheet 2

X	Y	Z
1	2	3
4	5	6

Table 6.6: Sheet 3

O	P	Q
3	2	1
4	3	2

Here is the code to obtain those sheets as a single dictionary:

```
>>> import pyexcel
>>> import json
>>> book_dict = pyexcel.get_book_dict(file_name="book.xls")
>>> isinstance(book_dict, OrderedDict)
True
>>> for key, item in book_dict.items():
...     print(json.dumps({key: item}))
{"Sheet 1": [[1, 2, 3], [4, 5, 6], [7, 8, 9]]}
{"Sheet 2": [["X", "Y", "Z"], [1, 2, 3], [4, 5, 6]]}
{"Sheet 3": [["O", "P", "Q"], [3, 2, 1], [4, 3, 2]]}
```

6.4.7 How to save a dictionary of two dimensional array as an excel file

Suppose you want to save the below dictionary to an excel file

```
>>> a_dictionary_of_two_dimensional_arrays = {
...      'Sheet 1':
...          [
...              [1.0, 2.0, 3.0],
...              [4.0, 5.0, 6.0],
...              [7.0, 8.0, 9.0]
...          ],
...      'Sheet 2':
...          [
...              ['X', 'Y', 'Z'],
...              [1.0, 2.0, 3.0],
...              [4.0, 5.0, 6.0]
...          ],
...      'Sheet 3':
...          [
...              ['O', 'P', 'Q'],
...              [3.0, 2.0, 1.0],
...              [4.0, 3.0, 2.0]
...          ]
...  }
```

Here is the code:

```
>>> pyexcel.save_book_as(
...    bookdict=a_dictionary_of_two_dimensional_arrays,
...    dest_file_name="book.xls"
... )
```

If you want to preserve the order of sheets in your dictionary, you have to pass on an ordered dictionary to the function itself. For example:

```
>>> data = OrderedDict()
>>> data.update({"Sheet 2": a_dictionary_of_two_dimensional_arrays['Sheet 2']})
>>> data.update({"Sheet 1": a_dictionary_of_two_dimensional_arrays['Sheet 1']})
>>> data.update({"Sheet 3": a_dictionary_of_two_dimensional_arrays['Sheet 3']})
>>> pyexcel.save_book_as(bookdict=data, dest_file_name="book.xls")
```

Let's verify its order:

```
>>> book_dict = pyexcel.get_book_dict(file_name="book.xls")
>>> for key, item in book_dict.items():
...     print(json.dumps({key: item}))
{"Sheet 2": [["X", "Y", "Z"], [1, 2, 3], [4, 5, 6]]}
{"Sheet 1": [[1, 2, 3], [4, 5, 6], [7, 8, 9]]}
{"Sheet 3": [["O", "P", "Q"], [3, 2, 1], [4, 3, 2]]}
```

Please notice that "Sheet 2" is the first item in the *book_dict*, meaning the order of sheets are preserved.

6.4.8 How to an excel sheet to a database using SQLAlchemy

Note: You can find the complete code of this example in examples folder on github

---

Before going ahead, let's import the needed components and initialize sql engine and table base:

```
>>> from sqlalchemy import create_engine
>>> from sqlalchemy.ext.declarative import declarative_base
>>> from sqlalchemy import Column , Integer, String, Float, Date
>>> from sqlalchemy.orm import sessionmaker
>>> engine = create_engine("sqlite:///birth.db")
>>> Base = declarative_base()
>>> Session = sessionmaker(bind=engine)
```

Let's suppose we have the following database model:

```
>>> class BirthRegister(Base):
...     __tablename__='birth'
...     id=Column(Integer, primary_key=True)
...     name=Column(String)
...     weight=Column(Float)
...     birth=Column(Date)
```

Let's create the table:

```
>>> Base.metadata.create_all(engine)
```

Now here is a sample excel file to be saved to the table:

name	weight	birth
Adam	3.4	2015-02-03
Smith	4.2	2014-11-12

Here is the code to import it:

```
>>> session = Session() # obtain a sql session
>>> pyexcel.save_as(file_name="birth.xls", name_columns_by_row=0, dest_session=session, dest_table=Bi
```

Done it. It is that simple. Let's verify what has been imported to make sure.

```
>>> sheet = pyexcel.get_sheet(session=session, table=BirthRegister)
>>> sheet
birth:
+------------+----+-------+--------+
| birth      | id | name  | weight |
+------------+----+-------+--------+
| 2015-02-03 | 1  | Adam  | 3.4    |
+------------+----+-------+--------+
| 2014-11-12 | 2  | Smith | 4.2    |
+------------+----+-------+--------+
```

6.4.9 How to open an xls file and save it as csv

Suppose we want to save previous used example 'birth.xls' as a csv file

```
>>> import pyexcel
>>> pyexcel.save_as(file_name="birth.xls", dest_file_name="birth.csv")
```

Again it is really simple. Let's verify what we have gotten:

```
>>> sheet = pyexcel.get_sheet(file_name="birth.csv")
>>> sheet
birth.csv:
+-------+--------+----------+
| name  | weight | birth    |
```

```
+-------+--------+----------+
| Adam  | 3.4    | 03/02/15 |
+-------+--------+----------+
| Smith | 4.2    | 12/11/14 |
+-------+--------+----------+
```

Note: Please note that csv(comma separate value) file is pure text file. Formula, charts, images and formatting in xls file will disappear no matter which transcoding tool you use. Hence, pyexcel is a quick alternative for this transcoding job.

6.4.10 How to open an xls file and save it as xlsx

Warning: Formula, charts, images and formatting in xls file will disappear as pyexcel does not support Formula, charts, images and formatting.

Let use previous example and save it as ods instead

```
>>> import pyexcel
>>> pyexcel.save_as(file_name="birth.xls",
...                 dest_file_name="birth.xlsx") # change the file extension
```

Again let's verify what we have gotten:

```
>>> sheet = pyexcel.get_sheet(file_name="birth.xlsx")
>>> sheet
pyexcel_sheet1:
+-------+--------+----------+
| name  | weight | birth    |
+-------+--------+----------+
| Adam  | 3.4    | 03/02/15 |
+-------+--------+----------+
| Smith | 4.2    | 12/11/14 |
+-------+--------+----------+
```

6.4.11 How to open a xls multiple sheet excel book and save it as csv

Well, you write similiar codes as before but you will need to use *:meth:~pyexcel.save_book_as* function.

6.5 Sheet: Formatting

Previous section has assumed the data is in the format that you want. In reality, you have to manipulate the data types a bit to suit your needs. Hence, formatters comes into the scene. The formatters take effect when the data is read on the fly. They do not affect the persistence of the data in the excel files. A row or column formatter can be applied to mutilpe rows/columns. There are two ways of applying a formatter:

1. use `add_formatter()`, `remove_formatter()` and `clear_formatter()` to apply formatter on the fly. The formatter takes effect when a cell value is read. In other words, the sheet content is intact until you call `freeze_formatters()` to apply all added formatters.
2. use `format()` to apply formatter immediately.

There is slightly different behavior between csv reader and xls reader. The cell type of the cells read by csv reader will be always text while the cell types read by xls reader vary.

6.5.1 Convert a column of numbers to strings

By default, all values in **csv** are read back as texts. However, for xls, xlsx and xlsm files, diffrent data type are supported. Numbers are always read as *float*. Therefore, if you should like to have them in string format, you need to do some conversions. Suppose you have the following data in any of the supported excel formats:

userid	name
10120	Adam
10121	Bella
10122	Cedar

Let's read it out first:

```
>>> import pyexcel
```

```
>>> sheet = pyexcel.get_sheet(file_name="example.xls", name_columns_by_row=0)
>>> sheet.column["userid"]
[10120, 10121, 10122]
```

As you can see, *userid* column is of *float* type. Next, let's convert the column to string format:

```
>>> sheet.column.format(0, str)
>>> sheet.column["userid"]
['10120', '10121', '10122']
```

Now, they are in string format.

You can do this row by row as well using `RowFormatter` or do this to a whote spread sheet using `SheetFormatter`

6.5.2 Cleanse the cells in a spread sheet

Sometimes, the data in a spreadsheet may have unwanted strings in all or some cells. Let's take an example. Suppose we have a spread sheet that contains all strings but it as random spaces before and after the text values. Some field had weird characters, such as " ":

Version	Comments	Author
v0.0.1	Release versions	Eda
v0.0.2	Useful updates	Freud

First, let's read the content and see what do we have:

```
>>> sheet = pyexcel.get_sheet(file_name="example.xls")
```

```
>>> sheet.to_array()
[['        Version', '        Comments', '        Author  '], ['  v0.0.1        ', ' Release vers
```

Now try to create a custom cleanse function:

```
>>> def cleanse_func(v):
...     v = v.replace(" ", "")
...     v = v.rstrip().strip()
...     return v
...
```

Then let's create a `SheetFormatter` and apply it:

```
>>> sf = pyexcel.formatters.SheetFormatter(cleanse_func)
>>> sheet.add_formatter(sf)
>>> sheet.to_array()
[['Version', 'Comments', 'Author'], ['v0.0.1', 'Release versions', 'Eda'], ['v0.0.2', 'Useful updates
```

So in the end, you get this:

Version	Comments	Author
v0.0.1	Release versions	Eda
v0.0.2	Useful updates	Freud

6.6 Book: Sheet operations

6.6.1 Access to individual sheets

You can access individual sheet of a book via attribute:

```
>>> book = pyexcel.get_book(file_name="book.xls")
>>> book.sheet3
sheet3:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 7 | 8 | 9 |
+---+---+---+
```

or via array notations:

```
>>> book["sheet 1"] # there is a space in the sheet name
sheet 1:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
```

6.6.2 Merge excel books

Suppose you have two excel books and each had three sheets. You can merge them and get a new book:

You also can merge indivdual sheets:

```
>>> book1 = pyexcel.get_book(file_name="book1.xls")
>>> book2 = pyexcel.get_book(file_name="book2.xlsx")
>>> merged_book = book1 + book2
>>> merged_book = book1["Sheet 1"] + book2["Sheet 2"]
>>> merged_book = book1["Sheet 1"] + book2
>>> merged_book = book1 + book2["Sheet 2"]
```

6.6.3 Manipulate individual sheets

merge sheets into a single sheet

Suppose you want to merge many csv files row by row into a new sheet.

```
>>> import pyexcel as pe
>>> import glob
>>> merged = pyexcel.Sheet()
>>> for file in glob.glob("*.csv"):
...     merged.row += pe.get_sheet(file_name=file)
>>> merged.save_as("merged.csv")
```

6.7 Work with excel files

Warning: The pyexcel DOES NOT consider Fonts, Styles, Formulas and Charts at all. When you load a stylish excel and update it, you definitely will lose all those.

6.7.1 Add a new row to an existing file

Suppose you have one data file as the following:

example.xls

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

And you want to add a new row:

12, 11, 10

Here is the code:

```
>>> import pyexcel as pe
>>> sheet = pe.get_sheet(file_name="example.xls")
>>> sheet.row += [12, 11, 10]
>>> sheet.save_as("new_example.xls")
>>> pe.get_sheet(file_name="new_example.xls")
pyexcel sheet:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+----------+----------+----------+
| 1        | 4        | 7        |
+----------+----------+----------+
| 2        | 5        | 8        |
+----------+----------+----------+
| 3        | 6        | 9        |
+----------+----------+----------+
| 12       | 11       | 10       |
+----------+----------+----------+
```

6.7.2 Update an existing row to an existing file

Suppose you want to update the last row of the example file as:

['N/A', 'N/A', 'N/A']

Here is the sample code:

```
>>> import pyexcel as pe
>>> sheet = pe.get_sheet(file_name="example.xls")
>>> sheet.row[3] = ['N/A', 'N/A', 'N/A']
>>> sheet.save_as("new_example1.xls")
>>> pe.get_sheet(file_name="new_example1.xls")
pyexcel sheet:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+----------+----------+----------+
| 1        | 4        | 7        |
+----------+----------+----------+
| 2        | 5        | 8        |
+----------+----------+----------+
| N/A      | N/A      | N/A      |
+----------+----------+----------+
```

6.7.3 Add a new column to an existing file

And you want to add a column instead:

["Column 4", 10, 11, 12]

Here is the code:

```
>>> import pyexcel as pe
>>> sheet = pe.get_sheet(file_name="example.xls")
>>> sheet.column += ["Column 4", 10, 11, 12]
>>> sheet.save_as("new_example2.xls")
>>> pe.get_sheet(file_name="new_example2.xls")
pyexcel sheet:
+----------+----------+----------+----------+
| Column 1 | Column 2 | Column 3 | Column 4 |
+----------+----------+----------+----------+
| 1        | 4        | 7        | 10       |
+----------+----------+----------+----------+
| 2        | 5        | 8        | 11       |
+----------+----------+----------+----------+
| 3        | 6        | 9        | 12       |
+----------+----------+----------+----------+
```

6.7.4 Update an existing column to an existing file

Again let's update "Column 3" with:

[100, 200, 300]

Here is the sample code:

```
>>> import pyexcel as pe
>>> sheet = pe.get_sheet(file_name="example.xls")
>>> sheet.column[2] = ["Column 3", 100, 200, 300]
>>> sheet.save_as("new_example3.xls")
>>> pe.get_sheet(file_name="new_example3.xls")
pyexcel sheet:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+----------+----------+----------+
| 1        | 4        | 100      |
+----------+----------+----------+
| 2        | 5        | 200      |
+----------+----------+----------+
| 3        | 6        | 300      |
+----------+----------+----------+
```

Alternatively, you could have done like this:

```
>>> import pyexcel as pe
>>> sheet = pe.get_sheet(file_name="example.xls", name_columns_by_row=0)
>>> sheet.column["Column 3"] = [100, 200, 300]
>>> sheet.save_as("new_example4.xls")
>>> pe.get_sheet(file_name="new_example4.xls")
pyexcel sheet:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+----------+----------+----------+
| 1        | 4        | 100      |
+----------+----------+----------+
| 2        | 5        | 200      |
+----------+----------+----------+
| 3        | 6        | 300      |
+----------+----------+----------+
```

How about the same alternative solution to previous row based example? Well, you'd better to have the following kind of data

row_example.xls

Row 1	1	2	3
Row 2	4	5	6
Row 3	7	8	9

And then you want to update "Row 3" with for example:

```
[100, 200, 300]
```

These code would do the job:

```
>>> import pyexcel as pe
>>> sheet = pe.get_sheet(file_name="row_example.xls", name_rows_by_column=0)
>>> sheet.row["Row 3"] = [100, 200, 300]
>>> sheet.save_as("new_example5.xls")
>>> pe.get_sheet(file_name="new_example5.xls")
pyexcel sheet:
+-------+-----+-----+-----+
| Row 1 | 1   | 2   | 3   |
+-------+-----+-----+-----+
| Row 2 | 4   | 5   | 6   |
+-------+-----+-----+-----+
```

```
| Row 3 | 100 | 200 | 300 |
+-------+-----+-----+-----+
```

6.8 Work with excel files in memory

Excel files in memory can be manipulated directly without saving it to physical disk and vice versa. This is useful in excel file handling at file upload or in excel file download. For example:

```
>>> import pyexcel

>>> content = "1,2,3\n3,4,5"
>>> sheet = pyexcel.get_sheet(file_type="csv", file_content=content)
>>> sheet.format(int)
>>> print(sheet.to_array())
[[1, 2, 3], [3, 4, 5]]
```

6.8.1 Read any supported excel and respond its content in json

You can find a real world example in **examples/memoryfile/** directory: pyexcel_server.py. Here is the example snippet

```
1  def upload():
2      if request.method == 'POST' and 'excel' in request.files:
3          # handle file upload
4          filename = request.files['excel'].filename
5          extension = filename.split(".")[1]
6          # Obtain the file extension and content
7          # pass a tuple instead of a file name
8          sheet = pyexcel.load_from_memory(extension, request.files['excel'].read())
9          # then use it as usual
10         data = pyexcel.to_dict(sheet)
11         # respond with a json
12         return jsonify({"result":data})
13     return render_template...
```

request.files['excel'] in line 4 holds the file object. line 5 finds out the file extension. line 8 feeds in a tuple to **Book**. line 10 gives a dictionary represetation of the excel file and line 11 send the json represetation of the excel file back to client browser

6.8.2 Write to memory and respond to download

```
1  data = [
2      [...],
3      ...
4  ]
5
6  @app.route('/download')
7  def download():
8      sheet = pe.Sheet(data)
9      io = StringIO()
10     sheet.save_to_memory("csv", io)
11     output = make_response(io.getvalue())
12     output.headers["Content-Disposition"] = "attachment; filename=export.csv"
```

```
    output.headers["Content-type"] = "text/csv"
    return output
```

make_response is a Flask utility to make a memory content as http response.

Note: You can find the corresponding source code at examples/memoryfile

Relevant packages

Readily made plugins have been made on top of this example. Here is a list of them:

framework	plugin/middleware/extension
Flask	Flask-Excel
Django	django-excel
Pyramid	pyramid-excel

And you may make your own by using pyexcel-webio

6.9 Migrate from 0.2.1 to 0.2.2

6.9.1 1. Explicit imports, no longer needed

Please forget about these statements:

```
import pyexcel.ext.xls
import pyexcel.ext.ods
import pyexcel.ext.xlsx
```

They are no longer needed. As long as you have pip-installed them, they will be auto-loaded. However, if you do not want some of the plugins, please use *pip* to uninstall them.

What if you have your code as it is? No harm but a few warnings shown:

```
Deprecated usage since v0.2.2! Explicit import is no longer required. pyexcel.ext.ods is auto importe
```

6.9.2 2. Invalid environment marker: platform_python_implementation=="PyPy"

Yes, it is a surprise. Please upgrade setuptools in your environment:

```
pip install --upgrade setuptools
```

At the time of writing, setuptools (18.0.1) or setuptools-21.0.0-py2.py3-none-any.whl is installed on author's computer and worked.

6.9.3 3. How to keep both pyexcel-xls and pyexcel-xlsx

As in Issue 20, pyexcel-xls was used for xls and pyexcel-xlsx had to be used for xlsx. Both must co-exist due to requirements. The workaround would failed when auto-import are enabled in v0.2.2. Hence, user of pyexcel in this situation shall use 'library' parameter to all signature functions, to instruct pyexcel to use a named library for each function call.

6.9.4 4. pyexcel.get_io is no longer exposed

pyexcel.get_io was passed on from pyexcel-io. However, it is no longer exposed. Please use pyexcel_io.manager.RWManager.get_io if you have to.

You are likely to use pyexcel.get_io when you do `pyexcel.Sheet.save_to_memory()` or `pyexcel.Book.save_to_memory()` where you need to put in a io stream. But actually, with latest code, you could put in a *None*.

6.10 Migrate from 0.1.x to 0.2.x

6.10.1 1. "Writer" is gone, Please use save_as.

Here is a piece of legacy code:

```
w = pyexcel.Writer("afile.csv")
data=[['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 1.1, 1]]
w.write_array(table)
w.close()
```

The new code is:

```
>>> data=[['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 1.1, 1]]
>>> pyexcel.save_as(array=data, dest_file_name="afile.csv")
```

Here is another piece of legacy code:

```
content = {
    "X": [1,2,3,4,5],
    "Y": [6,7,8,9,10],
    "Z": [11,12,13,14,15],
}
w = pyexcel.Writer("afile.csv")
w.write_dict(self.content)
w.close()
```

The new code is:

```
>>> content = {
...     "X": [1,2,3,4,5],
...     "Y": [6,7,8,9,10],
...     "Z": [11,12,13,14,15],
... }
>>> pyexcel.save_as(adict=content, dest_file_name="afile.csv")
```

Here is yet another piece of legacy code:

```
data = [
    [1, 2, 3],
    [4, 5, 6]
]
io = StringIO()
w = pyexcel.Writer(("csv",io))
w.write_rows(data)
w.close()
```

The new code is:

```
>>> data = [
...     [1, 2, 3],
...     [4, 5, 6]
... ]
>>> io = pyexcel.save_as(dest_file_type='csv', array=data)
>>> for line in io.readlines():
...     print(line.rstrip())
1,2,3
4,5,6
```

6.10.2 2. "BookWriter" is gone. Please use save_book_as.

Here is a piece of legacy code:

```
import pyexcel
content = {
         "Sheet1": [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3]],
         "Sheet2": [[4, 4, 4, 4], [5, 5, 5, 5], [6, 6, 6, 6]],
         "Sheet3": [[u'X', u'Y', u'Z'], [1, 4, 7], [2, 5, 8], [3, 6, 9]]
     }
w = pyexcel.BookWriter("afile.csv")
w.write_book_from_dict(content)
w.close()
```

The replacement code is:

```
>>> import pyexcel
>>> content = {
...          "Sheet1": [[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3]],
...          "Sheet2": [[4, 4, 4, 4], [5, 5, 5, 5], [6, 6, 6, 6]],
...          "Sheet3": [[u'X', u'Y', u'Z'], [1, 4, 7], [2, 5, 8], [3, 6, 9]]
...      }
>>> pyexcel.save_book_as(bookdict=content, dest_file_name="afile.csv")
```

CHAPTER 7

Cook book

7.1 Recipies

Warning: The pyexcel DOES NOT consider Fonts, Styles and Charts at all. In the resulting excel files, fonts, styles and charts will not be transferred.

These recipies give a one-stop utility functions for known use cases. Simliar functionality can be achieved using other application interfaces.

7.1.1 Update one column of a data file

Suppose you have one data file as the following:

example.xls

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

And you want to update `Column 2` with these data: [11, 12, 13]

Here is the code:

```
>>> from pyexcel.cookbook import update_columns
>>> custom_column = {"Column 2":[11, 12, 13]}
>>> update_columns("example.xls", custom_column, "output.xls")
```

Your output.xls will have these data:

Column 1	Column 2	Column 3
1	11	7
2	12	8
3	13	9

7.1.2 Update one row of a data file

Suppose you have the same data file:

example.xls

Row 1	1	2	3
Row 2	4	5	6
Row 3	7	8	9

And you want to update the second row with these data: [7, 4, 1]

Here is the code:

```
>>> from pyexcel.cookbook import update_rows
>>> custom_row = {"Row 1":[11, 12, 13]}
>>> update_rows("example.xls", custom_row, "output.xls")
```

Your output.xls will have these data:

Column 1	Column 2	Column 3
7	4	1
2	5	8
3	6	9

7.1.3 Merge two files into one

Suppose you want to merge the following two data files:

example.csv

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

example.xls

Column 4	Column 5
10	12
11	13

The following code will merge the tow into one file, say "output.xls":

```
>>> from pyexcel.cookbook import merge_two_files
>>> merge_two_files("example.csv", "example.xls", "output.xls")
```

The output.xls would have the following data:

Column 1	Column 2	Column 3	Column 4	Column 5
1	4	7	10	12
2	5	8	11	13
3	6	9		

7.1.4 Select candidate columns of two files and form a new one

Suppose you have these two files:

example.ods

Column 1	Column 2	Column 3	Column 4	Column 5
1	4	7	10	13
2	5	8	11	14
3	6	9	12	15

example.xls

Column 6	Column 7	Column 8	Column 9	Column 10
16	17	18	19	20

```
>>> data = [
...      ["Column 1", "Column 2", "Column 3", "Column 4", "Column 5"],
...      [1, 4, 7, 10, 13],
...      [2, 5, 8, 11, 14],
...      [3, 6, 9, 12, 15]
...  ]
>>> s = pyexcel.Sheet(data)
>>> s.save_as("example.csv")
>>> data = [
...      ["Column 6", "Column 7", "Column 8", "Column 9", "Column 10"],
...      [16, 17, 18, 19, 20]
...  ]
>>> s = pyexcel.Sheet(data)
>>> s.save_as("example.xls")
```

And you want to filter out column 2 and 4 from example.ods, filter out column 6 and 7 and merge them:

Column 1	Column 3	Column 5	Column 8	Column 9	Column 10
1	7	13	18	19	20
2	8	14			
3	9	15			

The following code will do the job:

```
>>> from pyexcel.cookbook import merge_two_readers
>>> from pyexcel.filters import EvenColumnFilter, ColumnFilter
>>> sheet1 = pyexcel.get_sheet(file_name="example.csv", name_columns_by_row=0)
>>> sheet2 = pyexcel.get_sheet(file_name="example.xls", name_columns_by_row=0)
>>> sheet1.filter(pyexcel.EvenColumnFilter())
>>> sheet2.filter(pyexcel.ColumnFilter([0, 1]))
>>> merge_two_readers(sheet1, sheet2, "output.xls")
```

7.1.5 Merge two files into a book where each file become a sheet

Suppose you want to merge the following two data files:

example.csv

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

example.xls

Column 4	Column 5
10	12
11	13

```
>>> data = [
...      ["Column 1", "Column 2", "Column 3"],
...      [1, 2, 3],
...      [4, 5, 6],
...      [7, 8, 9]
```

```
...     ]
>>> s = pyexcel.Sheet(data)
>>> s.save_as("example.csv")
>>> data = [
...         ["Column 4", "Column 5"],
...         [10, 12],
...         [11, 13]
...     ]
>>> s = pyexcel.Sheet(data)
>>> s.save_as("example.xls")
```

The following code will merge the tow into one file, say "output.xls":

```
>>> from pyexcel.cookbook import merge_all_to_a_book
>>> merge_all_to_a_book(["example.csv", "example.xls"], "output.xls")
```

The output.xls would have the following data:

example.csv as sheet name and inside the sheet, you have:

Column 1	Column 2	Column 3
1	4	7
2	5	8
3	6	9

example.ods as sheet name and inside the sheet, you have:

Column 4	Column 5
10	12
11	13

7.1.6 Merge all excel files in directory into a book where each file become a sheet

The following code will merge every excel files into one file, say "output.xls":

```
from pyexcel.cookbook import merge_all_to_a_book
import glob


merge_all_to_a_book(glob.glob("your_csv_directory\*.csv"), "output.xls")
```

You can mix and match with other excel formats: xls, xlsm and ods. For example, if you are sure you have only xls, xlsm, xlsx, ods and csv files in *your_excel_file_directory*, you can do the following:

```
from pyexcel.cookbook import merge_all_to_a_book
import glob


merge_all_to_a_book(glob.glob("your_excel_file_directory\*.*"), "output.xls")
```

7.1.7 Split a book into single sheet files

Suppose you have many sheets in a work book and you would like to separate each into a single sheet excel file. You can easily do this:

```
>>> from pyexcel.cookbook import split_a_book
>>> split_a_book("megabook.xls", "output.xls")
>>> import glob
>>> outputfiles = glob.glob("*_output.xls")
>>> for file in sorted(outputfiles):
...     print(file)
...
Sheet 1_output.xls
Sheet 2_output.xls
Sheet 3_output.xls
```

for the output file, you can specify any of the supported formats

7.1.8 Extract just one sheet from a book

Suppose you just want to extract one sheet from many sheets that exists in a work book and you would like to separate it into a single sheet excel file. You can easily do this:

```
>>> from pyexcel.cookbook import extract_a_sheet_from_a_book
>>> extract_a_sheet_from_a_book("megabook.xls", "Sheet 1", "output.xls")
>>> if os.path.exists("Sheet 1_output.xls"):
...     print("Sheet 1_output.xls exists")
...
Sheet 1_output.xls exists
```

for the output file, you can specify any of the supported formats

7.2 Loading from other sources

7.2.1 How to load a sheet from a url

Suppose you have excel file somewhere hosted:

```
>>> sheet = pe.get_sheet(url='http://yourdomain.com/test.csv')
>>> sheet
csv:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
```

CHAPTER 8

Real world cases

8.1 Questions and Answers

1. Python flask writing to a csv file and reading it
2. PyQt: Import .xls file and populate QTableWidget?
3. How do I write data to csv file in columns and rows from a list in python?
4. How to write dictionary values to a csv file using Python
5. How to select every Nth row in CSV file using python
6. Reading coordinates from text file in python
7. Python convert csv to xlsx
8. Python .xlsx (Office OpenXML) reader as simple as csv module?
9. Printing a specific line in CSV file with Python
10. Read .xls file data row by row python
11. How to read data from excel and set data type
12. Remove or keep specific columns in csv file
13. How can I put a CSV file in an array?
14. Python - write data into csv format as string (not file)

CHAPTER 9

API documentation

9.1 API Reference

This is intended for users of pyexcel.

Note: sphinx on ReadTheDocs cannot produce api docs. Please read it from [pypi](http://pythonhosted.org/pyexcel/api.html)

9.1.1 Signature functions

These flags can be passed on to control plugin behaviors:

auto_detect_int

Automatically convert float values to integers if the float number has no decimal values(e.g. 1.00). By default, it does the detection. Setting it to False will turn on this behavior

It has no effect on pyexcel-xlsx because it does that by default.

auto_detect_float

Automatically convert text to float values if possible. This applies only pyexcel-io where csv, tsv, csvz and tsvz formats are supported. By default, it does the detection. Setting it to False will turn on this behavior

auto_detect_datetime

Automatically convert text to python datetime if possible. This applies only pyexcel-io where csv, tsv, csvz and tsvz formats are supported. By default, it does the detection. Setting it to False will turn on this behavior

library

Name a pyexcel plugin to handle a file format. In the situation where multiple plugins were pip installed, it is confusing for pyexcel on which plugin to handle the file format. For example, both pyexcel-xlsx and pyexcel-xls reads xlsx format. Now since version 0.2.2, you can pass on *library="pyexcel-xls"* to handle xlsx in a specific function call.

Alternatively, you could uninstall the unwanted pyexcel plugin using pip.

Obtaining data from excel file

get_array(**keywords)	Obtain an array from an excel source
get_dict([name_columns_by_row])	Obtain a dictionary from an excel source
get_records([name_columns_by_row])	Obtain a list of records from an excel source
get_book_dict(**keywords)	Obtain a dictionary of two dimensional arrays
get_book(**keywords)	Get an instance of Book from an excel source
get_sheet(**keywords)	Get an instance of Sheet from an excel source

pyexcel.get_array

pyexcel.**get_array**(***keywords*)

Obtain an array from an excel source

Parameters **keywords** – see get_sheet()

pyexcel.get_dict

pyexcel.**get_dict**(*name_columns_by_row=0*, ***keywords*)

Obtain a dictionary from an excel source

Parameters

- **name_columns_by_row** – specify a row to be a dictionary key. It is default to 0 or first row.
- **keywords** – see get_sheet()

If you would use a column index 0 instead, you should do:

```
get_dict(name_columns_by_row=-1, name_rows_by_column=0)
```

pyexcel.get_records

pyexcel.**get_records**(*name_columns_by_row=0*, ***keywords*)

Obtain a list of records from an excel source

Parameters

- **name_columns_by_row** – specify a row to be a dictionary key. It is default to 0 or first row.
- **keywords** – see get_sheet()

If you would use a column index 0 instead, you should do:

```
get_records(name_columns_by_row=-1, name_rows_by_column=0)
```

pyexcel.get_book_dict

pyexcel.**get_book_dict**(***keywords*)

Obtain a dictionary of two dimensional arrays

Parameters **keywords** – see get_book()

pyexcel.get_book

`pyexcel.get_book(**keywords)`

Get an instance of `Book` from an excel source

Parameters

- **file_name** – a file with supported file extension
- **file_content** – the file content
- **file_stream** – the file stream
- **file_type** – the file type in *content*
- **session** – database session
- **tables** – a list of database table
- **models** – a list of django models
- **bookdict** – a dictionary of two dimensional arrays
- **url** – a download http url for your excel file

see also *A list of supported data structures*

Here is a table of parameters:

source	parameters
loading from file	file_name, keywords
loading from memory	file_type, content, keywords
loading from sql	session, tables
loading from django models	models
loading from dictionary	bookdict

Where the dictionary should have text as keys and two dimensional array as values.

pyexcel.get_sheet

`pyexcel.get_sheet(**keywords)`

Get an instance of `Sheet` from an excel source

Parameters

- **file_name** – a file with supported file extension
- **file_content** – the file content
- **file_stream** – the file stream
- **file_type** – the file type in *content*
- **session** – database session
- **table** – database table
- **model** – a django model
- **adict** – a dictionary of one dimensional arrays
- **url** – a download http url for your excel file
- **with_keys** – load with previous dictionary's keys, default is True

- **records** – a list of dictionaries that have the same keys
- **array** – a two dimensional array, a list of lists
- **keywords** – additional parameters, see `Sheet.__init__()`
- **sheet_name** – sheet name. if sheet_name is not given, the default sheet at index 0 is loaded

Not all parameters are needed. Here is a table

source	parameters
loading from file	file_name, sheet_name, keywords
loading from memory	file_type, content, sheet_name, keywords
loading from sql	session, table
loading from sql in django	model
loading from query sets	any query sets(sqlalchemy or django)
loading from dictionary	adict, with_keys
loading from records	records
loading from array	array

see also *A list of supported data structures*

Saving data to excel file

`save_as`(**keywords)	Save a sheet from a data srouce to another one
`save_book_as`(**keywords)	Save a book from a data source to another one

pyexcel.save_as

`pyexcel.save_as(**keywords)`

Save a sheet from a data srouce to another one

It accepts two sets of keywords. Why two sets? one set is source, the other set is destination. In order to distiguish the two sets, source set will be exactly the same as the ones for `pyexcel.get_sheet()`; destination set are exactly the same as the ones for `pyexcel.Sheet.save_as` but require a 'dest' prefix.

param keywords additional keywords can be found at `pyexcel.get_sheet()`

param dest_file_name another file name. **out_file** is deprecated though is still accepted.

param dest_file_type this is needed if you want to save to memory

param dest_session the target database session

param dest_table the target destination table

param dest_model the target django model

param dest_mapdict a mapping dictionary, see `pyexcel.Sheet.save_to_memory()`

param dest_initializer a custom initializer function for table or model

param dest_mapdict nominate headers

param dest_batch_size object creation batch size. it is Django specific

if csv file is destination format, python csv fmtparams are accepted

for example: dest_lineterminator will replace default '

' to the one you specified :returns: IO stream if saving to memory. None otherwise

Saving to source	parameters
file	dest_file_name, dest_sheet_name, keywords with prefix 'dest'
memory	dest_file_type, dest_content, dest_sheet_name, keywords with prefix 'dest'
sql	dest_session, table, dest_initializer, dest_mapdict
django model	dest_model, dest_initializer, dest_mapdict, dest_batch_size

pyexcel.save_book_as

pyexcel.**save_book_as**(***keywords*)

Save a book from a data source to another one

Parameters

- **dest_file_name** – another file name. **out_file** is deprecated though is still accepted.
- **dest_file_type** – this is needed if you want to save to memory
- **dest_session** – the target database session
- **dest_tables** – the list of target destination tables
- **dest_models** – the list of target destination django models
- **dest_mapdicts** – a list of mapping dictionaries
- **dest_initializers** – table initialization fuctions
- **dest_mapdicts** – to nominate a model or table fields. Optional
- **dest_batch_size** – batch creation size. Optional
- **keywords** – additional keywords can be found at *pyexcel.get_sheet()*

Returns IO stream if saving to memory. None otherwise

Saving to source	parameters
file	dest_file_name, dest_sheet_name, keywords with prefix 'dest'
memory	dest_file_type, dest_content, dest_sheet_name, keywords with prefix 'dest'
sql	dest_session, dest_tables, dest_table_init_func, dest_mapdict
django model	dest_models, dest_initializers, dest_mapdict, dest_batch_size

9.1.2 Cookbook

merge_csv_to_a_book(filelist[, outfilename])	merge a list of csv files into a excel book
merge_all_to_a_book(filelist[, outfilename])	merge a list of excel files into a excel book
split_a_book(file_name[, outfilename])	Split a file into separate sheets
extract_a_sheet_from_a_book(file_name, sheetname)	Extract a sheet from a excel book

pyexcel.merge_csv_to_a_book

pyexcel.**merge_csv_to_a_book**(*filelist*, *outfilename='merged.xls'*)

merge a list of csv files into a excel book

Parameters

- **filelist** (*list*) – a list of accessible file path

- **outfilename** (*str*) – save the sheet as

pyexcel.merge_all_to_a_book

pyexcel.**merge_all_to_a_book**(*filelist*, *outfilename='merged.xls'*)
merge a list of excel files into a excel book

Parameters

- **filelist** (*list*) – a list of accessible file path
- **outfilename** (*str*) – save the sheet as

pyexcel.split_a_book

pyexcel.**split_a_book**(*file_name*, *outfilename=None*)
Split a file into separate sheets

Parameters

- **file_name** (*str*) – an accessible file name
- **outfilename** (*str*) – save the sheets with file suffix

pyexcel.extract_a_sheet_from_a_book

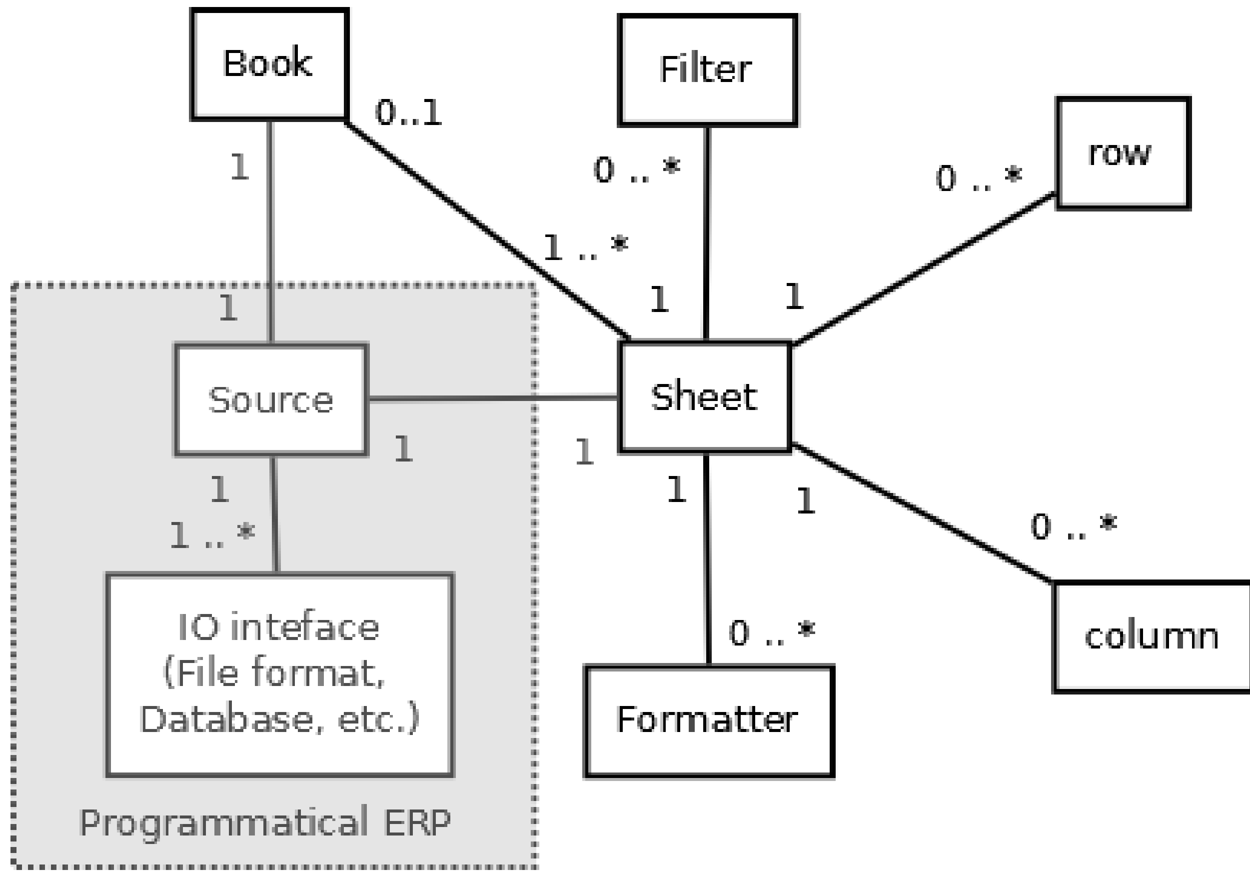
pyexcel.**extract_a_sheet_from_a_book**(*file_name*, *sheetname*, *outfilename=None*)
Extract a sheet from a excel book

Parameters

- **file_name** (*str*) – an accessible file name
- **sheetname** (*str*) – a valid sheet name
- **outfilename** (*str*) – save the sheet as

9.1.3 Book

Here's the entity relationship between Book, Sheet, Row and Column



Constructor

Book([sheets, filename, path])	Read an excel book that has one or more sheets

pyexcel.Book

class `pyexcel.`**`Book`**(*sheets=None*, *filename='memory'*, *path=None*)

Read an excel book that has one or more sheets

For csv file, there will be just one sheet

`__init__`(*sheets=None*, *filename='memory'*, *path=None*)

Book constructor

Selecting a specific book according to filename extension :param OrderedDict/dict sheets: a dictionary of data :param str filename: the physical file :param str path: the relative path or abosolute path :param set keywords: additional parameters to be passed on

Methods

__init__([sheets, filename, path])	Book constructor
`get_csv`(**keywords)	

Continued on next page

Table 9.5 – continued from previous page

get_csvz(**keywords)	
get_django(**keywords)	
Book.get_grid	
Book.get_html	
Book.get_json	
Book.get_latex	
Book.get_latex_booktabs	
Book.get_mediawiki	
get_ods(**keywords)	
Book.get_orgtbl	
Book.get_pipe	
Book.get_plain	
Book.get_rst	
get_sheet(array, name)	Create a sheet from a list of lists
Book.get_simple	
get_sql(**keywords)	
get_texttable(**keywords)	
get_tsv(**keywords)	
get_tsvz(**keywords)	
get_xls(**keywords)	
get_xlsm(**keywords)	
get_xlsx(**keywords)	
load_from_sheets(sheets)	Load content from existing sheets
number_of_sheets()	Return the number of sheets
register_presentation(file_type)	
remove_sheet(sheet)	Remove a sheet
save_as(filename)	Save the content to a new file
save_to(source)	Save to a writeable data source
save_to_database(session, tables[, ...])	Save data in sheets to database tables
save_to_django_models(models[, ...])	Save to database table through django model
save_to_memory(file_type[, stream])	Save the content to a memory stream
sheet_by_index(index)	Get the sheet with the specified index
sheet_by_name(name)	Get the sheet with the specified name
sheet_names()	Return all sheet names
to_dict()	Convert the book to a dictionary

Attributes

csv
csvz
django
Book.grid
Book.html
Book.json
Book.latex
Book.latex_booktabs
Book.mediawiki
ods
Book.orgtbl

Continued on next page

Table 9.6 – continued from previous page

Book.pipe
Book.plain
Book.rst
Book.simple
sql
texttable
tsv
tsvz
xls
xlsm
xlsx

Attribute

Book.number_of_sheets()	Return the number of sheets
Book.sheet_names()	Return all sheet names

pyexcel.Book.number_of_sheets

Book.**number_of_sheets**()
Return the number of sheets

pyexcel.Book.sheet_names

Book.**sheet_names**()
Return all sheet names

Conversions

Book.to_dict()	Convert the book to a dictionary

pyexcel.Book.to_dict

Book.**to_dict**()
Convert the book to a dictionary

Save changes

Book.save_to(source)	Save to a writeable data source
Book.save_as(filename)	Save the content to a new file
Book.save_to_memory(file_type[, stream])	Save the content to a memory stream
Book.save_to_database(session, tables[, ...])	Save data in sheets to database tables

pyexcel.Book.save_to

Book.**save_to**(*source*)
Save to a writeable data source

pyexcel.Book.save_as

Book.**save_as**(*filename*)
Save the content to a new file

Parameters **filename** (*str*) – a file path

pyexcel.Book.save_to_memory

Book.**save_to_memory**(*file_type*, *stream=None*, ***keywords*)
Save the content to a memory stream

Parameters

- **file_type** – what format the stream is in
- **stream** – a memory stream. Note in Python 3, for csv and tsv format, please pass an instance of StringIO. For xls, xlsx, and ods, an instance of BytesIO.

pyexcel.Book.save_to_database

Book.**save_to_database**(*session*, *tables*, *initializers=None*, *mapdicts=None*, *auto_commit=True*)
Save data in sheets to database tables

Parameters

- **session** – database session
- **tables** – a list of database tables, that is accepted by *Sheet.save_to_database()*. The sequence of tables matters when there is dependencies in between the tables. For example, **Car** is made by **Car Maker**. **Car Maker** table should be specified before **Car** table.
- **initializers** – a list of intialization functions for your tables and the sequence should match tables,
- **mapdicts** – custom map dictionary for your data columns and the sequence should match tables
- **auto_commit** – by default, data is committed.

9.1.4 Sheet

Constructor

Sheet([sheet, name, name_columns_by_row, ...])	Two dimensional data container for filtering, formatting and iteration

pyexcel.Sheet

class pyexcel.**Sheet**(*sheet=None*, *name='pyexcel sheet'*, *name_columns_by_row=-1*, *name_rows_by_column=-1*, *colnames=None*, *rownames=None*, *transpose_before=False*, *transpose_after=False*)

Two dimensional data container for filtering, formatting and iteration

Sheet is a container for a two dimensional array, where individual cell can be any Python types. Other than numbers, value of thsee types: string, date, time and boolean can be mixed in the array. This differs from Numpy's matrix where each cell are of the same number type.

In order to prepare two dimensional data for your computation, formatting functions help convert array cells to required types. Formatting can be applied not only to the whole sheet but also to selected rows or columns. Custom conversion function can be passed to these formatting functions. For example, to remove extra spaces surrounding the content of a cell, a custom function is required.

Filtering functions are used to reduce the information contained in the array.

__init__(*sheet=None*, *name='pyexcel sheet'*, *name_columns_by_row=-1*, *name_rows_by_column=-1*, *colnames=None*, *rownames=None*, *transpose_before=False*, *transpose_after=False*)

Constructor

Parameters

- **sheet** – two dimensional array
- **name** – this becomes the sheet name.
- **name_columns_by_row** – use a row to name all columns
- **name_rows_by_column** – use a column to name all rows
- **colnames** – use an external list of strings to name the columns
- **rownames** – use an external list of strings to name the rows

Methods

__init__([sheet, name, name_columns_by_row, ...])	Constructor
add_filter(afilter)	Apply a filter
add_formatter(aformatter)	Add a lazy formatter.
apply_formatter(aformatter)	Apply the formatter immediately.
cell_value(row, column[, new_value])	Random access to the data cells
clear_filters()	Clears all filters
clear_formatters()	Clear all formatters
column_at(index)	Gets the data at the specified column
column_range()	Utility function to get column range
columns()	Returns a left to right column iterator
contains(predicate)	Has something in the table
cut(topleft_corner, bottomright_corner)	Get a rectangle shaped data out and clear them in position
delete_columns(column_indices)	Delete one or more columns
delete_named_column_at(name)	Works only after you named columns by a row
delete_named_row_at(name)	Take the first column as row names
delete_rows(row_indices)	Delete one or more rows
enumerate()	Iterate cell by cell from top to bottom and from left to right
extend_columns(columns)	Take ordereddict to extend named columns

Continued on next page

Table 9.11 – continued from previous page

extend_columns_with_rows(rows)	Put rows on the right most side of the data
extend_rows(rows)	Take ordereddict to extend named rows
filter(afilter)	Apply the filter with immediate effect
format(formatter[, on_demand])	Apply a formatting action for the whole sheet
freeze_filters()	Apply all filters and delete them
freeze_formatters()	Apply all added formatters and clear them
get_csv(**keywords)	
get_csvz(**keywords)	
get_django(**keywords)	
Sheet.get_grid	
Sheet.get_html	
Sheet.get_json	
Sheet.get_latex	
Sheet.get_latex_booktabs	
Sheet.get_mediawiki	
get_ods(**keywords)	
Sheet.get_orgtbl	
Sheet.get_pipe	
Sheet.get_plain	
Sheet.get_rst	
Sheet.get_simple	
get_sql(**keywords)	
get_texttable(**keywords)	
get_tsv(**keywords)	
get_tsvz(**keywords)	
get_xls(**keywords)	
get_xlsm(**keywords)	
get_xlsx(**keywords)	
insert(topleft_corner[, rows, columns])	Insert a rectangle shaped data after a position
map(custom_function)	Execute a function across all cells of the sheet
name_columns_by_row(row_index)	Use the elements of a specified row to represent individual columns
name_rows_by_column(column_index)	Use the elements of a specified column to represent individual rows
named_column_at(name)	Get a column by its name
named_columns()	
named_row_at(name)	Get a row by its name
named_rows()	
number_of_columns()	Number of columns in the data sheet
number_of_rows()	Number of rows in the data sheet
paste(topleft_corner[, rows, columns])	Paste a rectangle shaped data after a position
rcolumns()	Returns a right to left column iterator
region(topleft_corner, bottomright_corner)	Get a rectangle shaped data out
register_presentation(file_type)	
remove_filter(afilter)	Remove a named filter
remove_formatter(aformatter)	Remove a formatter
reverse()	Opposite to enumerate
row_at(index)	Gets the data at the specified row
row_range()	Utility function to get row range
rows()	Returns a top to bottom row iterator
rrows()	Returns a bottom to top row iterator
rvertical()	Default iterator to go through each cell one by one from rightmost

Continued on next page

Table 9.11 – continued from previous page

save_as(filename, **keywords)	Save the content to a named file
save_to(source)	Save to a writeable data source
save_to_database(session, table[, ...])	Save data in sheet to database table
save_to_django_model(model[, initializer, ...])	Save to database table through django model
save_to_memory(file_type[, stream])	Save the content to memory
set_column_at(column_index, data_array[, ...])	Updates a column data range
set_named_column_at(name, column_array)	Take the first row as column names
set_named_row_at(name, row_array)	Take the first column as row names
set_row_at(row_index, data_array[, starting])	Update a row data range
to_array()	Returns an array after filtering
to_dict([row])	Returns a dictionary
to_records([custom_headers])	Returns the content as an array of dictionaries
transpose()	Roate the data table by 90 degrees
validate_filters()	Re-apply filters
vertical()	Default iterator to go through each cell one by one from

Attributes

colnames	Return column names
column	Column representation.
content	
csv	
csvz	
django	
Sheet.grid	
Sheet.html	
Sheet.json	
Sheet.latex	
Sheet.latex_booktabs	
Sheet.mediawiki	
ods	
Sheet.orgtbl	
Sheet.pipe	
Sheet.plain	
row	Row representation.
rownames	Return row names
Sheet.rst	
Sheet.simple	
sql	
texttable	
tsv	
tsvz	
xls	
xlsm	
xlsx	

Save changes

Sheet.save_to(source)	Save to a writeable data source
Sheet.save_as(filename, **keywords)	Save the content to a named file
Sheet.save_to_memory(file_type[, stream])	Save the content to memory
Sheet.save_to_database(session, table[, ...])	Save data in sheet to database table

pyexcel.Sheet.save_to

Sheet.**save_to**(*source*)
Save to a writeable data source

pyexcel.Sheet.save_as

Sheet.**save_as**(*filename*, ***keywords*)
Save the content to a named file

Heywords may vary depending on your file type, because the associated file type employs different library.

for csv, 'fmtparams <https://docs.python.org/release/3.1.5/ library/csv.html#dialects-and-formatting-parameters>'_ are accepted

for xls, 'auto_detect_int', 'encoding' and 'style_compression' are supported

for ods, 'auto_dtect_int' is supported

pyexcel.Sheet.save_to_memory

Sheet.**save_to_memory**(*file_type*, *stream=None*, ***keywords*)
Save the content to memory

Parameters

- **file_type** (*str*) – any value of 'csv', 'tsv', 'csvz', 'tsvz', 'xls', 'xlsm', 'xslm', 'ods'
- **stream** (*iostream*) – the memory stream to be written to. Note in Python 3, for csv and tsv format, please pass an instance of StringIO. For xls, xlsx, and ods, an instance of BytesIO.

pyexcel.Sheet.save_to_database

Sheet.**save_to_database**(*session*, *table*, *initializer=None*, *mapdict=None*, *auto_commit=True*)
Save data in sheet to database table

Parameters

- **session** – database session
- **table** – a database table
- **initializer** – a intialization functions for your table
- **mapdict** – custom map dictionary for your data columns
- **auto_commit** – by default, data is committed.

Attributes

Sheet.row	Row representation.
Sheet.column	Column representation.
Sheet.number_of_rows()	Number of rows in the data sheet
Sheet.number_of_columns()	Number of columns in the data sheet
Sheet.row_range()	Utility function to get row range
Sheet.column_range()	Utility function to get column range

pyexcel.Sheet.row

Sheet.**row**

Row representation. see NamedRow

examples:

```
>>> import pyexcel as pe
>>> data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> sheet = pe.Sheet(data)
>>> sheet.row[1]
[4, 5, 6]
>>> sheet.row[0:3]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> sheet.row += [11, 12, 13]
>>> sheet.row[3]
[11, 12, 13]
>>> sheet.row[0:4] = [0, 0, 0] # set all to zero
>>> sheet.row[3]
[0, 0, 0]
>>> sheet.row[0] = ['a', 'b', 'c'] # set one row
>>> sheet.row[0]
['a', 'b', 'c']
>>> del sheet.row[0] # delete first row
>>> sheet.row[0] # now, second row becomes the first
[0, 0, 0]
>>> del sheet.row[0:]
>>> sheet.row[0]  # nothing left
Traceback (most recent call last):
    ...
IndexError
```

pyexcel.Sheet.column

Sheet.**column**

Column representation. see NamedColumn

pyexcel.Sheet.number_of_rows

Sheet.**number_of_rows**()

Number of rows in the data sheet

pyexcel.Sheet.number_of_columns

Sheet.**number_of_columns**()
Number of columns in the data sheet

pyexcel.Sheet.row_range

Sheet.**row_range**()
Utility function to get row range

pyexcel.Sheet.column_range

Sheet.**column_range**()
Utility function to get column range

Iteration

Sheet.rows()	Returns a top to bottom row iterator
Sheet.rrows()	Returns a bottom to top row iterator
Sheet.columns()	Returns a left to right column iterator
Sheet.rcolumns()	Returns a right to left column iterator
Sheet.enumerate()	Iterate cell by cell from top to bottom and from left to right
Sheet.reverse()	Opposite to enumerate
Sheet.vertical()	Default iterator to go through each cell one by one from
Sheet.rvertical()	Default iterator to go through each cell one by one from rightmost

pyexcel.Sheet.rows

Sheet.**rows**()
Returns a top to bottom row iterator

example:

```
import pyexcel as pe
data = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
m = pe.Matrix(data)
print(pe.utils.to_array(m.rows()))
```

output:

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

More details see RowIterator

pyexcel.Sheet.rrows

`Sheet.rrows()`

Returns a bottom to top row iterator

```
import pyexcel as pe
data = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
m = pe.Matrix(data)
print(pe.utils.to_array(m.rrows()))
```

```
[[9, 10, 11, 12], [5, 6, 7, 8], [1, 2, 3, 4]]
```

More details see `RowReverseIterator`

pyexcel.Sheet.columns

`Sheet.columns()`

Returns a left to right column iterator

```
import pyexcel as pe
data = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
m = pe.Matrix(data)
print(pe.utils.to_array(m.columns()))
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

More details see `ColumnIterator`

pyexcel.Sheet.rcolumns

`Sheet.rcolumns()`

Returns a right to left column iterator

example:

```
import pyexcel as pe
data = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
m = pe.Matrix(data)
print(pe.utils.to_array(m.rcolumns()))
```

output:

```
[[4, 8, 12], [3, 7, 11], [2, 6, 10], [1, 5, 9]]
```

More details see `ColumnReverseIterator`

pyexcel.Sheet.enumerate

`Sheet.enumerate()`

Iterate cell by cell from top to bottom and from left to right

```
>>> import pyexcel as pe
>>> data = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]
>>> m = pe.sheets.Matrix(data)
>>> print(pe.to_array(m.enumerate()))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

More details see `HTLBRIterator`

pyexcel.Sheet.reverse

`Sheet.reverse()`

Opposite to enumerate

each cell one by one from bottom row to top row and from right to left example:

```
>>> import pyexcel as pe
>>> data = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]
>>> m = pe.sheets.Matrix(data)
>>> print(pe.to_array(m.reverse()))
[12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

More details see `HBRTLIterator`

pyexcel.Sheet.vertical

`Sheet.vertical()`

Default iterator to go through each cell one by one from leftmost column to rightmost row and from top to bottom example:

```
import pyexcel as pe
data = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
m = pe.Matrix(data)
print(pe.utils.to_array(m.vertical()))
```

output:

```
[1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12]
```

More details see `VTLBRIterator`

pyexcel.Sheet.rvertical

Sheet.**rvertical**()
Default iterator to go through each cell one by one from rightmost column to leftmost row and from bottom to top example:

```
import pyexcel as pe
data = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
m = pe.Matrix(data)
print(pe.utils.to_array(m.rvertical()))
```

output:

```
[12, 8, 4, 11, 7, 3, 10, 6, 2, 9, 5, 1]
```

More details see VBRTLIterator

Cell access

Sheet.cell_value(row, column[, new_value])	Random access to the data cells
Sheet.__getitem__(aset)	

pyexcel.Sheet.cell_value

Sheet.**cell_value**(*row*, *column*, *new_value=None*)
Random access to the data cells

pyexcel.Sheet.__getitem__

Sheet.**__getitem__**(*aset*)

Row access

Sheet.row_at(index)	Gets the data at the specified row
Sheet.set_row_at(row_index, data_array[, ...])	Update a row data range
Sheet.delete_rows(row_indices)	Delete one or more rows
Sheet.extend_rows(rows)	Take ordereddict to extend named rows

pyexcel.Sheet.row_at

Sheet.**row_at**(*index*)
Gets the data at the specified row

pyexcel.Sheet.set_row_at

`Sheet.set_row_at(row_index, data_array, starting=0)`
Update a row data range

It works like this if the call is: set_row_at(2, ['N', 'N', 'N'], 1):

```
A B C
1 3 5
2 N N <- row_index = 2
  ^starting = 1
```

This function will not set element outside the current table range

Parameters

- **row_index** (*int*) – which row to be modified
- **data_array** (*list*) – one dimensional array
- **starting** (*int*) – from which index, the update happens

Raises **IndexError** – if row_index exceeds row range or starting exceeds column range

pyexcel.Sheet.delete_rows

`Sheet.delete_rows(row_indices)`
Delete one or more rows

Parameters **row_indices** (*list*) – a list of row indices

pyexcel.Sheet.extend_rows

`Sheet.extend_rows(rows)`
Take ordereddict to extend named rows

Parameters **rows** (*ordereddist/list*) – a list of rows.

Column access

Sheet.column_at(index)	Gets the data at the specified column
Sheet.set_column_at(column_index, data_array)	Updates a column data range
Sheet.delete_columns(column_indices)	Delete one or more columns
Sheet.extend_columns(columns)	Take ordereddict to extend named columns

pyexcel.Sheet.column_at

`Sheet.column_at(index)`
Gets the data at the specified column

pyexcel.Sheet.set_column_at

Sheet.**set_column_at**(*column_index*, *data_array*, *starting=0*)
Updates a column data range

It works like this if the call is: set_column_at(2, ['N','N', 'N'], 1):

```
        +--> column_index = 2
        |
    A B C
    1 3 N <- starting = 1
    2 4 N
```

This function will not set element outside the current table range

Parameters

- **column_index** (*int*) – which column to be modified
- **data_array** (*list*) – one dimensional array
- **staring** (*int*) – from which index, the update happens

Raises IndexError – if column_index exceeds column range or starting exceeds row range

pyexcel.Sheet.delete_columns

Sheet.**delete_columns**(*column_indices*)
Delete one or more columns

Parameters column_indices (*list*) – a list of column indices

pyexcel.Sheet.extend_columns

Sheet.**extend_columns**(*columns*)
Take ordereddict to extend named columns

Parameters columns (*ordereddist/list*) – a list of columns

Data series

Any column as row name

Sheet.name_columns_by_row(row_index)	Use the elements of a specified row to represent individual columns
Sheet.rownames	Return row names
Sheet.named_column_at(name)	Get a column by its name
Sheet.set_named_column_at(name, column_array)	Take the first row as column names
Sheet.delete_named_column_at(name)	Works only after you named columns by a row

pyexcel.Sheet.name_columns_by_row

Sheet.**name_columns_by_row**(*row_index*)
Use the elements of a specified row to represent individual columns

The specified row will be deleted from the data :param int row_index: the index of the row that has the column names

pyexcel.Sheet.rownames

`Sheet.rownames`

Return row names

pyexcel.Sheet.named_column_at

`Sheet.named_column_at(name)`

Get a column by its name

pyexcel.Sheet.set_named_column_at

`Sheet.set_named_column_at(name, column_array)`

Take the first row as column names

Given name to identify the column index, set the column to the given array except the column name.

pyexcel.Sheet.delete_named_column_at

`Sheet.delete_named_column_at(name)`

Works only after you named columns by a row

Given name to identify the column index, set the column to the given array except the column name. :param str name: a column name

Any row as column name

`Sheet.name_rows_by_column`(column_index)	Use the elements of a specified column to represent individual rows
`Sheet.colnames`	Return column names
`Sheet.named_row_at`(name)	Get a row by its name
`Sheet.set_named_row_at`(name, row_array)	Take the first column as row names
`Sheet.delete_named_row_at`(name)	Take the first column as row names

pyexcel.Sheet.name_rows_by_column

`Sheet.name_rows_by_column(column_index)`

Use the elements of a specified column to represent individual rows

The specified column will be deleted from the data :param int column_index: the index of the column that has the row names

pyexcel.Sheet.colnames

`Sheet.colnames`

Return column names

pyexcel.Sheet.named_row_at

`Sheet.named_row_at(name)`

Get a row by its name

pyexcel.Sheet.set_named_row_at

`Sheet.set_named_row_at(name, row_array)`

Take the first column as row names

Given name to identify the row index, set the row to the given array except the row name.

pyexcel.Sheet.delete_named_row_at

`Sheet.delete_named_row_at(name)`

Take the first column as row names

Given name to identify the row index, set the row to the given array except the row name.

Formatting

Sheet.format(formatter[, on_demand])	Apply a formatting action for the whole sheet
Sheet.apply_formatter(aformatter)	Apply the formatter immediately.
Sheet.add_formatter(aformatter)	Add a lazy formatter.
Sheet.remove_formatter(aformatter)	Remove a formatter
Sheet.clear_formatters()	Clear all formatters
Sheet.freeze_formatters()	Apply all added formatters and clear them

pyexcel.Sheet.format

`Sheet.format(formatter, on_demand=False)`

Apply a formatting action for the whole sheet

Example:

```
>>> import pyexcel as pe
>>> # Given a dictinoary as the following
>>> data = {
...     "1": [1, 2, 3, 4, 5, 6, 7, 8],
...     "3": [1.25, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8],
...     "5": [2, 3, 4, 5, 6, 7, 8, 9],
...     "7": [1, '',]
...     }
>>> sheet = pe.get_sheet(adict=data)
>>> sheet.row[1]
[1, 1.25, 2, 1]
>>> sheet.format(str)
>>> sheet.row[1]
['1', '1.25', '2', '1']
>>> sheet.format(int)
>>> sheet.row[1]
[1, 1, 2, 1]
```

pyexcel.Sheet.apply_formatter

`Sheet.apply_formatter(aformatter)`

Apply the formatter immediately.

Parameters **aformatter** (*Formatter*) – a custom formatter

pyexcel.Sheet.add_formatter

`Sheet.add_formatter(aformatter)`

Add a lazy formatter.

The formatter takes effect on the fly when a cell value is read This is cost effective when you have a big data table and you use only a few rows or columns. If you have farily modest data table, you can choose apply_formatter() too.

Parameters **aformatter** (*Formatter*) – a custom formatter

pyexcel.Sheet.remove_formatter

Sheet.**remove_formatter**(*aformatter*)

Remove a formatter

Parameters **aformatter** (*Formatter*) – a custom formatter

pyexcel.Sheet.clear_formatters

Sheet.**clear_formatters**()

Clear all formatters

Example:

```
>>> import pyexcel as pe
>>> # Given a dictinoary as the following
>>> data = {
...     "1": [1, 2, 3, 4, 5, 6, 7, 8],
...     "3": [1.25, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8],
...     "5": [2, 3, 4, 5, 6, 7, 8, 9],
...     "7": [1, '',]
...     }
>>> sheet = pe.get_sheet(adict=data)
>>> sheet.row[1]
[1, 1.25, 2, 1]
>>> inc = lambda value: (float(value) if value != None else 0)+1
>>> aformatter = pe.SheetFormatter(inc)
>>> sheet.add_formatter(aformatter)
>>> sheet.row[1]
[2.0, 2.25, 3.0, 2.0]
>>> sheet.clear_formatters()
>>> sheet.row[1]
[1, 1.25, 2, 1]
```

pyexcel.Sheet.freeze_formatters

Sheet.**freeze_formatters**()

Apply all added formatters and clear them

The tradeoff here is when you extend the sheet, you won't get the effect of previously applied formatters because they are applied and gone.

Filtering

Sheet.filter(afilter)	Apply the filter with immediate effect
Sheet.add_filter(afilter)	Apply a filter

Continued on next page

Table 9.22 – continued from previous page

Sheet.remove_filter(afilter)	Remove a named filter
Sheet.clear_filters()	Clears all filters
Sheet.freeze_filters()	Apply all filters and delete them

pyexcel.Sheet.filter

Sheet.**filter**(*afilter*)
Apply the filter with immediate effect

pyexcel.Sheet.add_filter

Sheet.**add_filter**(*afilter*)
Apply a filter

Parameters **afilter** (*Filter*) – a custom filter

pyexcel.Sheet.remove_filter

Sheet.**remove_filter**(*afilter*)
Remove a named filter

have to remove all filters in order to re-validate the rest of the filters

pyexcel.Sheet.clear_filters

Sheet.**clear_filters**()
Clears all filters

pyexcel.Sheet.freeze_filters

Sheet.**freeze_filters**()
Apply all filters and delete them

Conversion

Sheet.to_array()	Returns an array after filtering
Sheet.to_dict([row])	Returns a dictionary
Sheet.to_records([custom_headers])	Returns the content as an array of dictionaries

pyexcel.Sheet.to_array

Sheet.**to_array**()
Returns an array after filtering

pyexcel.Sheet.to_dict

Sheet.**to_dict**(*row=False*)
Returns a dictionary

pyexcel.Sheet.to_records

Sheet.**to_records**(*custom_headers=None*)
Returns the content as an array of dictionaries

Anti-conversion

dict_to_array(*arg, **keywords)
from_records(*arg, **keywords)

pyexcel.dict_to_array

pyexcel.**dict_to_array**(**arg, **keywords*)

pyexcel.from_records

pyexcel.**from_records**(**arg, **keywords*)

Transformation

Sheet.transpose()	Roate the data table by 90 degrees
Sheet.map(custom_function)	Execute a function across all cells of the sheet
Sheet.region(topleft_corner, bottomright_corner)	Get a rectangle shaped data out
Sheet.cut(topleft_corner, bottomright_corner)	Get a rectangle shaped data out and clear them in position
Sheet.paste(topleft_corner[, rows, columns])	Paste a rectangle shaped data after a position

pyexcel.Sheet.transpose

Sheet.**transpose**()
Roate the data table by 90 degrees

Reference *transpose()*

pyexcel.Sheet.map

Sheet.**map**(*custom_function*)
Execute a function across all cells of the sheet

Example:

```
>>> import pyexcel as pe
>>> # Given a dictinoary as the following
```

```
>>> data = {
...     "1": [1, 2, 3, 4, 5, 6, 7, 8],
...     "3": [1.25, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8],
...     "5": [2, 3, 4, 5, 6, 7, 8, 9],
...     "7": [1, '',]
...     }
>>> sheet = pe.get_sheet(adict=data)
>>> sheet.row[1]
[1, 1.25, 2, 1]
>>> inc = lambda value: (float(value) if value != None else 0)+1
>>> sheet.map(inc)
>>> sheet.row[1]
[2.0, 2.25, 3.0, 2.0]
```

pyexcel.Sheet.region

Sheet.**region**(*topleft_corner*, *bottomright_corner*)
Get a rectangle shaped data out

Parameters

- **topleft_corner** (*slice*) – the top left corner of the rectangle
- **bottomright_corner** (*slice*) – the bottom right corner of the rectangle

example:

```
>>> import pyexcel as pe
>>> data = [
...     # 0 1  2  3  4 5   6
...     [1, 2, 3, 4, 5, 6, 7], #  0
...     [21, 22, 23, 24, 25, 26, 27],
...     [31, 32, 33, 34, 35, 36, 37],
...     [41, 42, 43, 44, 45, 46, 47],
...     [51, 52, 53, 54, 55, 56, 57]  # 4
... ]
>>> s = pe.Sheet(data)
>>> data = s.cut([1, 1], [4, 5])
>>> s2 = pe.Sheet(data) #  let's present the result
>>> s2
pyexcel sheet:
+----+----+----+----+
| 22 | 23 | 24 | 25 |
+----+----+----+----+
| 32 | 33 | 34 | 35 |
+----+----+----+----+
| 42 | 43 | 44 | 45 |
+----+----+----+----+
```

pyexcel.Sheet.cut

Sheet.**cut**(*topleft_corner*, *bottomright_corner*)
Get a rectangle shaped data out and clear them in position

Parameters

- **topleft_corner** (*slice*) – the top left corner of the rectangle

- **bottomright_corner** (*slice*) – the bottom right corner of the rectangle

example:

```
>>> import pyexcel as pe
>>> data = [
...     # 0 1  2  3  4 5   6
...     [1, 2, 3, 4, 5, 6, 7], #  0
...     [21, 22, 23, 24, 25, 26, 27],
...     [31, 32, 33, 34, 35, 36, 37],
...     [41, 42, 43, 44, 45, 46, 47],
...     [51, 52, 53, 54, 55, 56, 57]  # 4
... ]
>>> s = pe.Sheet(data)
>>> s
pyexcel sheet:
+----+----+----+----+----+----+----+
| 1  | 2  | 3  | 4  | 5  | 6  | 7  |
+----+----+----+----+----+----+----+
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
+----+----+----+----+----+----+----+
| 31 | 32 | 33 | 34 | 35 | 36 | 37 |
+----+----+----+----+----+----+----+
| 41 | 42 | 43 | 44 | 45 | 46 | 47 |
+----+----+----+----+----+----+----+
| 51 | 52 | 53 | 54 | 55 | 56 | 57 |
+----+----+----+----+----+----+----+
>>> # cut  1<= row < 4, 1<= column < 5
>>> data = s.cut([1, 1], [4, 5])
>>> s
pyexcel sheet:
+----+----+----+----+----+----+----+
| 1  | 2  | 3  | 4  | 5  | 6  | 7  |
+----+----+----+----+----+----+----+
| 21 |    |    |    |    | 26 | 27 |
+----+----+----+----+----+----+----+
| 31 |    |    |    |    | 36 | 37 |
+----+----+----+----+----+----+----+
| 41 |    |    |    |    | 46 | 47 |
+----+----+----+----+----+----+----+
| 51 | 52 | 53 | 54 | 55 | 56 | 57 |
+----+----+----+----+----+----+----+
```

pyexcel.Sheet.paste

Sheet.**paste**(*topleft_corner*, *rows=None*, *columns=None*)

Paste a rectangle shaped data after a position

Parameters **topleft_corner** (*slice*) – the top left corner of the rectangle

example:

```
>>> import pyexcel as pe
>>> data = [
...     # 0 1  2  3  4 5   6
...     [1, 2, 3, 4, 5, 6, 7], #  0
...     [21, 22, 23, 24, 25, 26, 27],
...     [31, 32, 33, 34, 35, 36, 37],
...     [41, 42, 43, 44, 45, 46, 47],
```

```
...     [51, 52, 53, 54, 55, 56, 57]  # 4
... ]
>>> s = pe.Sheet(data)
>>> # cut  1<= row < 4, 1<= column < 5
>>> data = s.cut([1, 1], [4, 5])
>>> s.paste([4,6], rows=data)
>>> s
pyexcel sheet:
+----+----+----+----+----+----+----+----+----+----+
| 1  | 2  | 3  | 4  | 5  | 6  | 7  |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
| 21 |    |    |    |    | 26 | 27 |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
| 31 |    |    |    |    | 36 | 37 |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
| 41 |    |    |    |    | 46 | 47 |    |    |    |
+----+----+----+----+----+----+----+----+----+----+
| 51 | 52 | 53 | 54 | 55 | 56 | 22 | 23 | 24 | 25 |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    | 32 | 33 | 34 | 35 |
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    | 42 | 43 | 44 | 45 |
+----+----+----+----+----+----+----+----+----+----+
>>> s.paste([6,9], columns=data)
>>> s
pyexcel sheet:
+----+----+----+----+----+----+----+----+----+----+----+----+
| 1  | 2  | 3  | 4  | 5  | 6  | 7  |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+
| 21 |    |    |    |    | 26 | 27 |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+
| 31 |    |    |    |    | 36 | 37 |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+
| 41 |    |    |    |    | 46 | 47 |    |    |    |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+
| 51 | 52 | 53 | 54 | 55 | 56 | 22 | 23 | 24 | 25 |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    | 32 | 33 | 34 | 35 |    |    |
+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    | 42 | 43 | 44 | 22 | 32 | 42 |
+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    | 23 | 33 | 43 |
+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    | 24 | 34 | 44 |
+----+----+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    |    |    |    | 25 | 35 | 45 |
+----+----+----+----+----+----+----+----+----+----+----+----+
```

9.1.5 Row access

NamedRow(matrix)	Series Sheet would have Named Row instead of Row
NamedRow.format([row_index, formatter, ...])	Format a row
NamedRow.select(names)	Delete row indices other than specified

pyexcel.sheets.NamedRow

class pyexcel.sheets.**NamedRow**(*matrix*)

Series Sheet would have Named Row instead of Row

Here is an example to merge sheets. Suppose we have the following three files:

```
>>> import pyexcel as pe
>>> data = [[1,2,3],[4,5,6],[7,8,9]]
>>> s = pe.Sheet(data)
>>> s.save_as("1.csv")
>>> data2 = [['a','b','c'],['d','e','f'],['g','h','i']]
>>> s2 = pe.Sheet(data2)
>>> s2.save_as("2.csv")
>>> data3=[[1.1, 2.2, 3.3],[4.4, 5.5, 6.6],[7.7, 8.8, 9.9]]
>>> s3=pe.Sheet(data3)
>>> s3.save_as("3.csv")


>>> merged = pe.Sheet()
>>> for file in ["1.csv", "2.csv", "3.csv"]:
...     r = pe.get_sheet(file_name=file)
...     merged.row += r
>>> merged.save_as("merged.csv")
```

Now let's verify what we had:

```
>>> sheet = pe.get_sheet(file_name="merged.csv")
```

this is added to overcome doctest's inability to handle python 3's unicode:

```
>>> sheet.format(lambda v: str(v))
>>> sheet
merged.csv:
+-----+-----+-----+
| 1   | 2   | 3   |
+-----+-----+-----+
| 4   | 5   | 6   |
+-----+-----+-----+
| 7   | 8   | 9   |
+-----+-----+-----+
| a   | b   | c   |
+-----+-----+-----+
| d   | e   | f   |
+-----+-----+-----+
| g   | h   | i   |
+-----+-----+-----+
| 1.1 | 2.2 | 3.3 |
+-----+-----+-----+
| 4.4 | 5.5 | 6.6 |
+-----+-----+-----+
| 7.7 | 8.8 | 9.9 |
+-----+-----+-----+
```

__init__(*matrix*)

Methods

`__init__`(matrix)	
`format`([row_index, formatter, format_specs, ...])	Format a row
`select`(names)	Delete row indices other than specified

pyexcel.sheets.NamedRow.format

`NamedRow.format` (*row_index=None*, *formatter=None*, *format_specs=None*, *on_demand=False*)

Format a row

pyexcel.sheets.NamedRow.select

`NamedRow.select` (*names*)

Delete row indices other than specified

Examples:

```
>>> import pyexcel as pe
>>> data = [[1],[2],[3],[4],[5],[6],[7],[9]]
>>> sheet = pe.Sheet(data)
>>> sheet
pyexcel sheet:
+---+
| 1 |
+---+
| 2 |
+---+
| 3 |
+---+
| 4 |
+---+
| 5 |
+---+
| 6 |
+---+
| 7 |
+---+
| 9 |
+---+
>>> sheet.row.select([1,2,3,5])
>>> sheet
pyexcel sheet:
+---+
| 2 |
+---+
| 3 |
+---+
| 4 |
+---+
| 6 |
+---+
>>> data = [
...     ['a', 1],
...     ['b', 1],
...     ['c', 1]
... ]
>>> sheet = pe.Sheet(data, name_rows_by_column=0)
```

```
>>> sheet.row.select(['a', 'b'])
>>> sheet
pyexcel sheet:
+---+---+
| a | 1 |
+---+---+
| b | 1 |
+---+---+
```

9.1.6 Column access

NamedColumn(matrix)	Series Sheet would have Named Column instead of Column
NamedColumn.format([column_index, ...])	Format a column
NamedColumn.select(names)	Delete columns other than specified

pyexcel.sheets.NamedColumn

class pyexcel.sheets.**NamedColumn**(*matrix*)

Series Sheet would have Named Column instead of Column

example:

```
import pyexcel as pe

r = pe.SeriesReader("example.csv")
print(r.column["column 1"])
```

__init__(*matrix*)

Methods

__init__(matrix)	
format([column_index, formatter, ...])	Format a column
select(names)	Delete columns other than specified

pyexcel.sheets.NamedColumn.format

NamedColumn.**format**(*column_index=None*, *formatter=None*, *format_specs=None*, *on_demand=False*)

Format a column

pyexcel.sheets.NamedColumn.select

NamedColumn.**select**(*names*)

Delete columns other than specified

Examples:

```
>>> import pyexcel as pe
>>> data = [[1,2,3,4,5,6,7,9]]
>>> sheet = pe.Sheet(data)
>>> sheet
```

```
pyexcel sheet:
+---+---+---+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
+---+---+---+---+---+---+---+---+
>>> sheet.column.select([1,2,3,5])
>>> sheet
pyexcel sheet:
+---+---+---+---+
| 2 | 3 | 4 | 6 |
+---+---+---+---+
>>> data = [
...     ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'],
...     [1,2,3,4,5,6,7,9],
... ]
>>> sheet = pe.Sheet(data, name_columns_by_row=0)
>>> sheet
pyexcel sheet:
+---+---+---+---+---+---+---+---+
| a | b | c | d | e | f | g | h |
+===+===+===+===+===+===+===+===+
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
+---+---+---+---+---+---+---+---+
>>> del sheet.column['a', 'b', 'i', 'f']
Traceback (most recent call last):
    ...
ValueError: ...
>>> sheet.column.select(['a', 'c', 'e', 'h'])
>>> sheet
pyexcel sheet:
+---+---+---+---+
| a | c | e | h |
+===+===+===+===+
| 1 | 3 | 5 | 9 |
+---+---+---+---+
```

9.1.7 Data formatters

`ColumnFormatter`(column_index, formatter)	Apply formatting on columns
`NamedColumnFormatter`(column_index, formatter)	Apply formatting using named columns
`RowFormatter`(row_index, formatter)	Row Formatter
`NamedRowFormatter`(row_index, formatter)	Formatting rows using named rows
`SheetFormatter`(formatter)	Apply the formatter to all cells in the sheet

pyexcel.formatters.ColumnFormatter

class `pyexcel.formatters.`**`ColumnFormatter`**(*column_index*, *formatter*)

Apply formatting on columns

`__init__`(*column_index*, *formatter*)

Constructor :param column_index: to which column or what

columns to apply the formatter

Parameters **`formatter`** – the target format, or a custom functional formatter

Methods

`__init__`(column_index, formatter)	Constructor
`do_format`(value)	
`is_my_business`(row, column, value)	Check should this formatter be active for cell at (row, column) with value

pyexcel.formatters.NamedColumnFormatter

class `pyexcel.formatters.`**`NamedColumnFormatter`**(*column_index*, *formatter*)

Apply formatting using named columns

`__init__`(*column_index*, *formatter*)

Constructor :param column_index: to which column or what

columns to apply the formatter

Parameters **`formatter`** – the target format, or a custom functional formatter

Methods

`__init__`(column_index, formatter)	Constructor
`do_format`(value)	
`is_my_business`(row, column, value)	Check should this formatter be active for cell at (row, column) with value
`update_index`(new_indices)	

pyexcel.formatters.RowFormatter

class `pyexcel.formatters.`**`RowFormatter`**(*row_index*, *formatter*)

Row Formatter

`__init__`(*row_index*, *formatter*)

Constructor :param row_index: to which row or what

rows to apply the formatter

Parameters **`formatter`** – the target format, or a custom functional formatter

Methods

`__init__`(row_index, formatter)	Constructor
`do_format`(value)	
`is_my_business`(row, column, value)	Check should this formatter be active for cell at (row, column) with value

pyexcel.formatters.NamedRowFormatter

class `pyexcel.formatters.`**`NamedRowFormatter`**(*row_index*, *formatter*)

Formatting rows using named rows

`__init__`(*row_index*, *formatter*)

Constructor :param row_index: to which row or what

rows to apply the formatter

Parameters **formatter** – the target format, or a custom functional formatter

Methods

__init__(row_index, formatter)	Constructor
do_format(value)	
is_my_business(row, column, value)	Check should this formatter be active for cell at (row, column) with value
update_index(new_indices)	

pyexcel.formatters.SheetFormatter

class pyexcel.formatters.**SheetFormatter**(*formatter*)

Apply the formatter to all cells in the sheet

__init__(*formatter*)

Methods

__init__(formatter)
do_format(value)
is_my_business(row, column, value)

9.1.8 Data Filters

ColumnFilter(indices)	Filters out a list of columns
SingleColumnFilter(index)	Filters out a single column index
OddColumnFilter()	Filters out odd indexed columns
EvenColumnFilter()	Filters out even indexed columns
ColumnValueFilter(func)	Filters out rows based on its row values
RowFilter(indices)	Filters a list of rows
SingleRowFilter(index)	Filters out a single row
OddRowFilter()	Filters out odd indexed rows
EvenRowFilter()	Filters out even indexed rows
RowValueFilter(func)	Filters out rows based on its row values
RegionFilter(row_slice, column_slice)	Filter on both row index and column index

pyexcel.filters.ColumnFilter

class pyexcel.filters.**ColumnFilter**(*indices*)

Filters out a list of columns

__init__(*indices*)

Constructor

Parameters **indices** (*list*) – a list of column indices to be filtered out

Methods

`__init__`(indices)	Constructor
`columns`()	Columns that were filtered out
`invert`()	
`rows`()	Rows that were filtered out
`translate`(row, column)	Map the row, column after filtering to the
`validate_filter`(reader)	Find out which column index to be filtered

pyexcel.filters.SingleColumnFilter

class `pyexcel.filters.`**`SingleColumnFilter`**(*index*)

Filters out a single column index

`__init__`(*index*)

Constructor

Parameters **`indices`** (`list`) – a list of column indices to be filtered out

Methods

`__init__`(index)	Constructor
`columns`()	Columns that were filtered out
`invert`()	
`rows`()	Rows that were filtered out
`translate`(row, column)	Map the row, column after filtering to the
`validate_filter`(reader)	Find out which column index to be filtered

pyexcel.filters.OddColumnFilter

class `pyexcel.filters.`**`OddColumnFilter`**

Filters out odd indexed columns

- column 0 is regarded as the first column.
- column 1 is regarded as the seocond column -> this will be filtered out

`__init__`()

Methods

`__init__`()	
`columns`()	Columns that were filtered out
`invert`()	
`rows`()	Rows that were filtered out
`translate`(row, column)	Map the row, column after filtering to the
`validate_filter`(reader)	Find out which column index to be filtered

pyexcel.filters.EvenColumnFilter

class pyexcel.filters.**EvenColumnFilter**
Filters out even indexed columns

- column 0 is regarded as the first column. -> this will be filtered out
- column 1 is regarded as the seocond column

__init__()

Methods

__init__()	
columns()	Columns that were filtered out
invert()	
rows()	Rows that were filtered out
translate(row, column)	Map the row, column after filtering to the
validate_filter(reader)	Find out which column index to be filtered

pyexcel.filters.ColumnValueFilter

class pyexcel.filters.**ColumnValueFilter**(*func*)
Filters out rows based on its row values

Note: it takes time as it needs to go through all values

__init__(*func*)
Constructor :param Function func: a evaluation function

Methods

__init__(func)	Constructor
columns()	Columns that were filtered out
invert()	
rows()	Rows that were filtered out
translate(row, column)	Map the row, column after filtering to the
validate_filter(reader)	Filter out the row indices

pyexcel.filters.RowFilter

class pyexcel.filters.**RowFilter**(*indices*)
Filters a list of rows

__init__(*indices*)
Constructor

Parameters **indices** (*list*) – a list of column indices to be filtered out

Methods

`__init__`(indices)	Constructor
`columns`()	Columns that were filtered out
`invert`()	
`rows`()	number of rows to be filtered out
`translate`(row, column)	Map the row, column after filtering to the
`validate_filter`(reader)	Find out which column index to be filtered

pyexcel.filters.SingleRowFilter

class `pyexcel.filters.`**`SingleRowFilter`**(*index*)

Filters out a single row

`__init__`(*index*)

Constructor

Parameters **`indices`** (*`list`*) – a list of column indices to be filtered out

Methods

`__init__`(index)	Constructor
`columns`()	Columns that were filtered out
`invert`()	
`rows`()	number of rows to be filtered out
`translate`(row, column)	Map the row, column after filtering to the
`validate_filter`(reader)	Find out which column index to be filtered

pyexcel.filters.OddRowFilter

class `pyexcel.filters.`**`OddRowFilter`**

Filters out odd indexed rows

row 0 is seen as the first row

`__init__`()

Methods

`__init__`()	
`columns`()	Columns that were filtered out
`invert`()	
`rows`()	number of rows to be filtered out
`translate`(row, column)	Map the row, column after filtering to the
`validate_filter`(reader)	Find out which column index to be filtered

pyexcel.filters.EvenRowFilter

class pyexcel.filters.**EvenRowFilter**

Filters out even indexed rows

row 0 is seen as the first row

__init__()

Methods

__init__()	
columns()	Columns that were filtered out
invert()	
rows()	number of rows to be filtered out
translate(row, column)	Map the row, column after filtering to the
validate_filter(reader)	Find out which column index to be filtered

pyexcel.filters.RowValueFilter

class pyexcel.filters.**RowValueFilter**(*func*)

Filters out rows based on its row values

Note: it takes time as it needs to go through all values

__init__(*func*)

Constructor :param Function func: a evaluation function

Methods

__init__(func)	Constructor
columns()	Columns that were filtered out
invert()	
rows()	number of rows to be filtered out
translate(row, column)	Map the row, column after filtering to the
validate_filter(reader)	Filter out the row indices

pyexcel.filters.RegionFilter

class pyexcel.filters.**RegionFilter**(*row_slice*, *column_slice*)

Filter on both row index and column index

__init__(*row_slice*, *column_slice*)

Constructor

Parameters

- **row_slice** (slice) – row index range
- **column_slice** (slice) – column index range

Methods

`__init__`(row_slice, column_slice)	Constructor
`columns`()	Columns that were filtered out
`invert`()	
`rows`()	Rows that were filtered out
`translate`(row, column)	Map the row, column after filtering to the
`validate_filter`(reader)	

9.2 Internal API reference

This is intended for developers and hackers of pyexcel.

9.2.1 Data sheet representation

In inheritance order from parent to child

`Matrix`(array)	The internal representation of a sheet data.

pyexcel.sheets.Matrix

class `pyexcel.sheets.`**`Matrix`**(*array*)

The internal representation of a sheet data. Each element can be of any python types

`__init__`(*array*)

Constructor

The reason a deep copy was not made here is because the data sheet could be huge. It could be costly to copy every cell to a new memory area :param list array: a list of arrays

Methods

`__init__`(array)	Constructor
`cell_value`(row, column[, new_value])	Random access to table cells
`column_at`(index)	Gets the data at the specified column
`column_range`()	Utility function to get column range
`columns`()	Returns a left to right column iterator
`contains`(predicate)	Has something in the table
`delete_columns`(column_indices)	Delete columns by specified list of indices
`delete_rows`(row_indices)	Deletes specified row indices
`enumerate`()	Iterate cell by cell from top to bottom and from left to right
`extend_columns`(columns)	Inserts two dimensional data after the rightmost column
`extend_columns_with_rows`(rows)	Rows were appended to the rightmost side
`extend_rows`(rows)	Inserts two dimensinal data after the bottom row
`number_of_columns`()	The number of columns
`number_of_rows`()	The number of rows
`paste`(topleft_corner[, rows, columns])	Paste a rectangle shaped data after a position

Continued on next page

Table 9.49 – continued from previous page

rcolumns()	Returns a right to left column iterator
reverse()	Opposite to enumerate
row_at(index)	Gets the data at the specified row
row_range()	Utility function to get row range
rows()	Returns a top to bottom row iterator
rrows()	Returns a bottom to top row iterator
rvertical()	Default iterator to go through each cell one by one from rightmost
set_column_at(column_index, data_array[, ...])	Updates a column data range
set_row_at(row_index, data_array[, starting])	Update a row data range
to_array()	Get an array out
transpose()	Roate the data table by 90 degrees
vertical()	Default iterator to go through each cell one by one from

Attributes

column
row

FormattableSheet(array)	A represetation of Matrix that accept custom formatters
FilterableSheet(sheet)	A represetation of Matrix that can be filtered
NominableSheet([sheet, name, ...])	Allow dictionary group of the content
Sheet([sheet, name, name_columns_by_row, ...])	Two dimensional data container for filtering, formatting and iteration

pyexcel.sheets.FormattableSheet

class pyexcel.sheets.**FormattableSheet**(*array*)

A represetation of Matrix that accept custom formatters

__init__(*array*)

Constructor

Methods

__init__(array)	Constructor
add_formatter(aformatter)	Add a lazy formatter.
apply_formatter(aformatter)	Apply the formatter immediately.
cell_value(row, column[, new_value])	Random access to the data cells
clear_formatters()	Clear all formatters
column_at(index)	Gets the data at the specified column
column_range()	Utility function to get column range
columns()	Returns a left to right column iterator
contains(predicate)	Has something in the table
delete_columns(column_indices)	Delete columns by specified list of indices
delete_rows(row_indices)	Deletes specified row indices
enumerate()	Iterate cell by cell from top to bottom and from left to right
extend_columns(columns)	Inserts two dimensional data after the rightmost column

Continued on next page

Table 9.52 – continued from previous page

extend_columns_with_rows(rows)	Rows were appended to the rightmost side
extend_rows(rows)	Inserts two dimensinal data after the bottom row
format(formatter[, on_demand])	Apply a formatting action for the whole sheet
freeze_formatters()	Apply all added formatters and clear them
map(custom_function)	Execute a function across all cells of the sheet
number_of_columns()	The number of columns
number_of_rows()	The number of rows
paste(topleft_corner[, rows, columns])	Paste a rectangle shaped data after a position
rcolumns()	Returns a right to left column iterator
remove_formatter(aformatter)	Remove a formatter
reverse()	Opposite to enumerate
row_at(index)	Gets the data at the specified row
row_range()	Utility function to get row range
rows()	Returns a top to bottom row iterator
rrows()	Returns a bottom to top row iterator
rvertical()	Default iterator to go through each cell one by one from rightmost
set_column_at(column_index, data_array[, ...])	Updates a column data range
set_row_at(row_index, data_array[, starting])	Update a row data range
to_array()	Get an array out
transpose()	Roate the data table by 90 degrees
vertical()	Default iterator to go through each cell one by one from

Attributes

column
row

pyexcel.sheets.FilterableSheet

class pyexcel.sheets.**FilterableSheet**(*sheet*)

A represetation of Matrix that can be filtered by as many filters as it is applied

__init__(*sheet*)

Methods

__init__(sheet)	
add_filter(afilter)	Apply a filter
add_formatter(aformatter)	Add a lazy formatter.
apply_formatter(aformatter)	Apply the formatter immediately.
cell_value(row, column[, new_value])	Random access to the data cells
clear_filters()	Clears all filters
clear_formatters()	Clear all formatters
column_at(index)	Gets the data at the specified column
column_range()	Utility function to get column range
columns()	Returns a left to right column iterator
contains(predicate)	Has something in the table

Continued on next page

Table 9.54 – continued from previous page

`cut`(topleft_corner, bottomright_corner)	Get a rectangle shaped data out and clear them in position
`delete_columns`(*args)	
`delete_rows`(*args)	
`enumerate`()	Iterate cell by cell from top to bottom and from left to right
`extend_columns`(*args)	
`extend_columns_with_rows`(rows)	Rows were appended to the rightmost side
`extend_rows`(*args)	
`filter`(afilter)	Apply the filter with immediate effect
`format`(formatter[, on_demand])	Apply a formatting action for the whole sheet
`freeze_filters`()	Apply all filters and delete them
`freeze_formatters`()	Apply all added formatters and clear them
`insert`(topleft_corner[, rows, columns])	Insert a rectangle shaped data after a position
`map`(custom_function)	Execute a function across all cells of the sheet
`number_of_columns`()	Number of columns in the data sheet
`number_of_rows`()	Number of rows in the data sheet
`paste`(topleft_corner[, rows, columns])	Paste a rectangle shaped data after a position
`rcolumns`()	Returns a right to left column iterator
`region`(topleft_corner, bottomright_corner)	Get a rectangle shaped data out
`remove_filter`(afilter)	Remove a named filter
`remove_formatter`(aformatter)	Remove a formatter
`reverse`()	Opposite to enumerate
`row_at`(index)	Gets the data at the specified row
`row_range`()	Utility function to get row range
`rows`()	Returns a top to bottom row iterator
`rrows`()	Returns a bottom to top row iterator
`rvertical`()	Default iterator to go through each cell one by one from rightmost
`set_column_at`(column_index, data_array[, ...])	Updates a column data range
`set_row_at`(row_index, data_array[, starting])	Update a row data range
`to_array`()	Get an array out
`transpose`()	Roate the data table by 90 degrees
`validate_filters`()	Re-apply filters
`vertical`()	Default iterator to go through each cell one by one from

Attributes

`column`
`row`

pyexcel.sheets.NominableSheet

class `pyexcel.sheets.`**`NominableSheet`**(*sheet=None*, *name='pyexcel sheet'*, *name_columns_by_row=-1*, *name_rows_by_column=-1*, *colnames=None*, *rownames=None*, *transpose_before=False*, *transpose_after=False*)

Allow dictionary group of the content

`__init__`(*sheet=None*, *name='pyexcel sheet'*, *name_columns_by_row=-1*, *name_rows_by_column=-1*, *colnames=None*, *rownames=None*, *transpose_before=False*, *transpose_after=False*)

Constructor

Parameters

- **sheet** – two dimensional array
- **name** – this becomes the sheet name.
- **name_columns_by_row** – use a row to name all columns
- **name_rows_by_column** – use a column to name all rows
- **colnames** – use an external list of strings to name the columns
- **rownames** – use an external list of strings to name the rows

Methods

`__init__`([sheet, name, name_columns_by_row, ...])	Constructor
`add_filter`(afilter)	Apply a filter
`add_formatter`(aformatter)	Add a lazy formatter.
`apply_formatter`(aformatter)	Apply the formatter immediately.
`cell_value`(row, column[, new_value])	Random access to the data cells
`clear_filters`()	Clears all filters
`clear_formatters`()	Clear all formatters
`column_at`(index)	Gets the data at the specified column
`column_range`()	Utility function to get column range
`columns`()	Returns a left to right column iterator
`contains`(predicate)	Has something in the table
`cut`(topleft_corner, bottomright_corner)	Get a rectangle shaped data out and clear them in position
`delete_columns`(column_indices)	Delete one or more columns
`delete_named_column_at`(name)	Works only after you named columns by a row
`delete_named_row_at`(name)	Take the first column as row names
`delete_rows`(row_indices)	Delete one or more rows
`enumerate`()	Iterate cell by cell from top to bottom and from left to right
`extend_columns`(columns)	Take ordereddict to extend named columns
`extend_columns_with_rows`(rows)	Put rows on the right most side of the data
`extend_rows`(rows)	Take ordereddict to extend named rows
`filter`(afilter)	Apply the filter with immediate effect
`format`(formatter[, on_demand])	Apply a formatting action for the whole sheet
`freeze_filters`()	Apply all filters and delete them
`freeze_formatters`()	Apply all added formatters and clear them
`insert`(topleft_corner[, rows, columns])	Insert a rectangle shaped data after a position
`map`(custom_function)	Execute a function across all cells of the sheet
`name_columns_by_row`(row_index)	Use the elements of a specified row to represent individual columns
`name_rows_by_column`(column_index)	Use the elements of a specified column to represent individual rows
`named_column_at`(name)	Get a column by its name
`named_columns`()	
`named_row_at`(name)	Get a row by its name
`named_rows`()	
`number_of_columns`()	Number of columns in the data sheet
`number_of_rows`()	Number of rows in the data sheet
`paste`(topleft_corner[, rows, columns])	Paste a rectangle shaped data after a position
`rcolumns`()	Returns a right to left column iterator
`region`(topleft_corner, bottomright_corner)	Get a rectangle shaped data out
`remove_filter`(afilter)	Remove a named filter

Continued on next page

Table 9.56 – continued from previous page

`remove_formatter`(aformatter)	Remove a formatter
`reverse`()	Opposite to enumerate
`row_at`(index)	Gets the data at the specified row
`row_range`()	Utility function to get row range
`rows`()	Returns a top to bottom row iterator
`rrows`()	Returns a bottom to top row iterator
`rvertical`()	Default iterator to go through each cell one by one from rightmost
`set_column_at`(column_index, data_array[, ...])	Updates a column data range
`set_named_column_at`(name, column_array)	Take the first row as column names
`set_named_row_at`(name, row_array)	Take the first column as row names
`set_row_at`(row_index, data_array[, starting])	Update a row data range
`to_array`()	Returns an array after filtering
`to_dict`([row])	Returns a dictionary
`to_records`([custom_headers])	Returns the content as an array of dictionaries
`transpose`()	Roate the data table by 90 degrees
`validate_filters`()	Re-apply filters
`vertical`()	Default iterator to go through each cell one by one from

Attributes

`colnames`	Return column names
`column`	Column representation.
`row`	Row representation.
`rownames`	Return row names

pyexcel.sheets.Sheet

class `pyexcel.sheets.`**`Sheet`**(*sheet=None*, *name='pyexcel sheet'*, *name_columns_by_row=-1*, *name_rows_by_column=-1*, *colnames=None*, *rownames=None*, *transpose_before=False*, *transpose_after=False*)

Two dimensional data container for filtering, formatting and iteration

Sheet is a container for a two dimensional array, where individual cell can be any Python types. Other than numbers, value of thsee types: string, date, time and boolean can be mixed in the array. This differs from Numpy's matrix where each cell are of the same number type.

In order to prepare two dimensional data for your computation, formatting functions help convert array cells to required types. Formatting can be applied not only to the whole sheet but also to selected rows or columns. Custom conversion function can be passed to these formatting functions. For example, to remove extra spaces surrounding the content of a cell, a custom function is required.

Filtering functions are used to reduce the information contained in the array.

`__init__`(*sheet=None*, *name='pyexcel sheet'*, *name_columns_by_row=-1*, *name_rows_by_column=-1*, *colnames=None*, *rownames=None*, *transpose_before=False*, *transpose_after=False*)

Constructor

Parameters

- **`sheet`** – two dimensional array
- **`name`** – this becomes the sheet name.
- **`name_columns_by_row`** – use a row to name all columns

- **name_rows_by_column** – use a column to name all rows
- **colnames** – use an external list of strings to name the columns
- **rownames** – use an external list of strings to name the rows

Methods

`__init__`([sheet, name, name_columns_by_row, ...])	Constructor
`add_filter`(afilter)	Apply a filter
`add_formatter`(aformatter)	Add a lazy formatter.
`apply_formatter`(aformatter)	Apply the formatter immediately.
`cell_value`(row, column[, new_value])	Random access to the data cells
`clear_filters`()	Clears all filters
`clear_formatters`()	Clear all formatters
`column_at`(index)	Gets the data at the specified column
`column_range`()	Utility function to get column range
`columns`()	Returns a left to right column iterator
`contains`(predicate)	Has something in the table
`cut`(topleft_corner, bottomright_corner)	Get a rectangle shaped data out and clear them in position
`delete_columns`(column_indices)	Delete one or more columns
`delete_named_column_at`(name)	Works only after you named columns by a row
`delete_named_row_at`(name)	Take the first column as row names
`delete_rows`(row_indices)	Delete one or more rows
`enumerate`()	Iterate cell by cell from top to bottom and from left to right
`extend_columns`(columns)	Take ordereddict to extend named columns
`extend_columns_with_rows`(rows)	Put rows on the right most side of the data
`extend_rows`(rows)	Take ordereddict to extend named rows
`filter`(afilter)	Apply the filter with immediate effect
`format`(formatter[, on_demand])	Apply a formatting action for the whole sheet
`freeze_filters`()	Apply all filters and delete them
`freeze_formatters`()	Apply all added formatters and clear them
`get_csv`(**keywords)	
`get_csvz`(**keywords)	
`get_django`(**keywords)	
`Sheet.get_grid`	
`Sheet.get_html`	
`Sheet.get_json`	
`Sheet.get_latex`	
`Sheet.get_latex_booktabs`	
`Sheet.get_mediawiki`	
`get_ods`(**keywords)	
`Sheet.get_orgtbl`	
`Sheet.get_pipe`	
`Sheet.get_plain`	
`Sheet.get_rst`	
`Sheet.get_simple`	
`get_sql`(**keywords)	
`get_texttable`(**keywords)	
`get_tsv`(**keywords)	
`get_tsvz`(**keywords)	

Continued on next page

Table 9.58 – continued from previous page

get_xls(**keywords)	
get_xlsm(**keywords)	
get_xlsx(**keywords)	
insert(topleft_corner[, rows, columns])	Insert a rectangle shaped data after a position
map(custom_function)	Execute a function across all cells of the sheet
name_columns_by_row(row_index)	Use the elements of a specified row to represent individual columns
name_rows_by_column(column_index)	Use the elements of a specified column to represent individual rows
named_column_at(name)	Get a column by its name
named_columns()	
named_row_at(name)	Get a row by its name
named_rows()	
number_of_columns()	Number of columns in the data sheet
number_of_rows()	Number of rows in the data sheet
paste(topleft_corner[, rows, columns])	Paste a rectangle shaped data after a position
rcolumns()	Returns a right to left column iterator
region(topleft_corner, bottomright_corner)	Get a rectangle shaped data out
register_presentation(file_type)	
remove_filter(afilter)	Remove a named filter
remove_formatter(aformatter)	Remove a formatter
reverse()	Opposite to enumerate
row_at(index)	Gets the data at the specified row
row_range()	Utility function to get row range
rows()	Returns a top to bottom row iterator
rrows()	Returns a bottom to top row iterator
rvertical()	Default iterator to go through each cell one by one from rightmost
save_as(filename, **keywords)	Save the content to a named file
save_to(source)	Save to a writeable data source
save_to_database(session, table[, ...])	Save data in sheet to database table
save_to_django_model(model[, initializer, ...])	Save to database table through django model
save_to_memory(file_type[, stream])	Save the content to memory
set_column_at(column_index, data_array[, ...])	Updates a column data range
set_named_column_at(name, column_array)	Take the first row as column names
set_named_row_at(name, row_array)	Take the first column as row names
set_row_at(row_index, data_array[, starting])	Update a row data range
to_array()	Returns an array after filtering
to_dict([row])	Returns a dictionary
to_records([custom_headers])	Returns the content as an array of dictionaries
transpose()	Roate the data table by 90 degrees
validate_filters()	Re-apply filters
vertical()	Default iterator to go through each cell one by one from

Attributes

colnames	Return column names
column	Column representation.
content	
csv	
csvz	
django	

Continued on next page

Table 9.59 – continued from previous page

Sheet.grid	
Sheet.html	
Sheet.json	
Sheet.latex	
Sheet.latex_booktabs	
Sheet.mediawiki	
ods	
Sheet.orgtbl	
Sheet.pipe	
Sheet.plain	
row	Row representation.
rownames	Return row names
Sheet.rst	
Sheet.simple	
sql	
texttable	
tsv	
tsvz	
xls	
xlsm	
xlsx	

9.2.2 Row represetation

Row(matrix)	Represet row of a matrix

pyexcel.sheets.Row

class pyexcel.sheets.**Row**(*matrix*)

Represet row of a matrix

Table 9.61: "example.csv"

1	2	3
4	5	6
7	8	9

Above column manipluation can be performed on rows similiarly. This section will not repeat the same example but show some advance usages.

```
>>> import pyexcel as pe
>>> data = [[1,2,3], [4,5,6], [7,8,9]]
>>> m = pe.sheets.Matrix(data)
>>> m.row[0:2]
[[1, 2, 3], [4, 5, 6]]
>>> m.row[0:3] = [0, 0, 0]
>>> m.row[2]
[0, 0, 0]
>>> del m.row[0:2]
>>> m.row[0]
[0, 0, 0]
```

`__init__(matrix)`

Methods

`__init__`(matrix)	
`select`(indices)	Delete row indices other than specified

9.2.3 Column represetation

`Column`(matrix)	Represet columns of a matrix

pyexcel.sheets.Column

class `pyexcel.sheets.Column`(*matrix*)

Represet columns of a matrix

Table 9.64: "example.csv"

1	2	3
4	5	6
7	8	9

Let us manipulate the data columns on the above data matrix:

```
>>> import pyexcel as pe
>>> data = [[1,2,3], [4,5,6], [7,8,9]]
>>> m = pe.sheets.Matrix(data)
>>> m.column[0]
[1, 4, 7]
>>> m.column[2] = [0, 0, 0]
>>> m.column[2]
[0, 0, 0]
>>> del m.column[1]
>>> m.column[1]
[0, 0, 0]
>>> m.column[2]
Traceback (most recent call last):
    ...
IndexError
```

`__init__(matrix)`

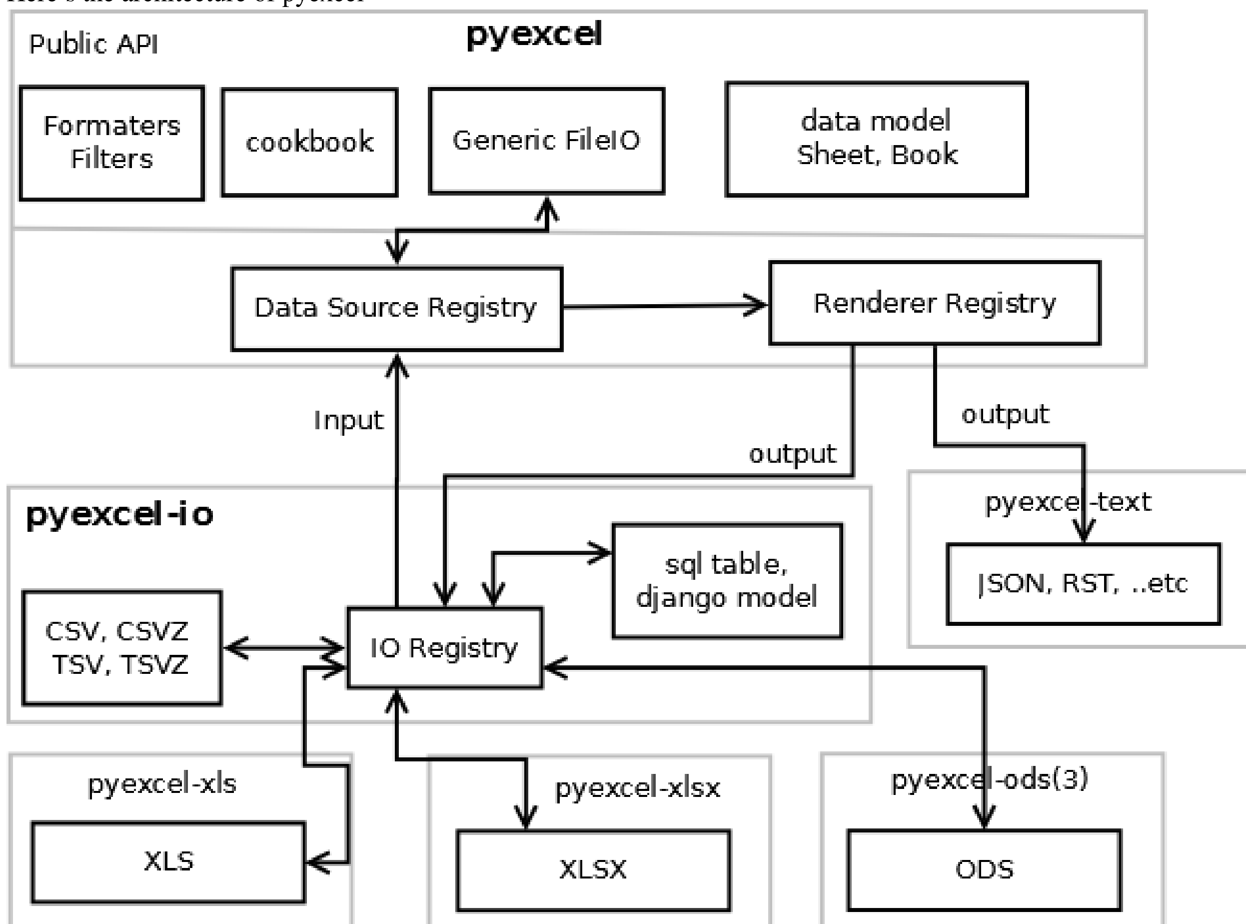
Methods

`__init__`(matrix)	
`select`(indices)	Examples:

Developer's guide

10.1 Developer's guide

Here's the architecture of pyexcel



Pull requests are welcome.

Development steps for code changes

1. git clone https://github.com/pyexcel/pyexcel.git
2. cd pyexcel

Upgrade your setup tools and pip. They are needed for development and testing only:

1. pip install –upgrade setuptools "pip==7.1"

Then install relevant development requirements:

1. pip install -r rnd_requirements.txt # if such a file exists
2. pip install -r requirements.txt
3. pip install -r tests/requirements.txt

In order to update test environment, and documentation, additional setps are required:

1. pip install moban
2. git clone https://github.com/pyexcel/pyexcel-commons.git
3. make your changes in *.moban.d* directory, then issue command *moban*

10.1.1 What is rnd_requirements.txt

Usually, it is created when a dependent library is not released. Once the dependecy is installed(will be released), the future version of the dependency in the requirements.txt will be valid.

10.1.2 What is pyexcel-commons

Many information that are shared across pyexcel projects, such as: this developer guide, license info, etc. are stored in *pyexcel-commons* project.

10.1.3 What is .moban.d

.moban.d stores the specific meta data for the library.

10.1.4 How to test your contribution

Although *nose* and *doctest* are both used in code testing, it is adviable that unit tests are put in tests. *doctest* is incorporated only to make sure the code examples in documentation remain valid across different development releases.

On Linux/Unix systems, please launch your tests like this:

```
$ make test
```

On Windows systems, please issue this command:

```
> test.bat
```

10.1.5 Acceptance criteria

1. Has fair amount of documentation
2. Has Test cases written
3. Has all code lines tested
4. Passes all Travis CI builds

5. Pythonic code please
6. Agree on NEW BSD License for your contribution

CHAPTER 11

Change log

11.1 Change log

11.1.1 0.2.5 - 31.08.2016

Updated:

1. # 58: texttable should have been made as compulsory requirement

11.1.2 0.2.4 - 14.07.2016

Updated:

1. For python 2, writing to sys.stdout by pyexcel-cli raise IOError.

11.1.3 0.2.3 - 11.07.2016

Updated:

1. For python 3, do not seek 0 when saving to memory if sys.stdout is passed on. Hence, adding support for sys.stdin and sys.stdout.

11.1.4 0.2.2 - 01.06.2016

Updated:

1. Explicit imports, no longer needed
2. Depends on latest setuptools 18.0.1
3. NotImplementedError will be raised if parameters to core functions are not supported, e.g. get_sheet(cannot_find_me_option="will be thrown out as NotImplementedError")

11.1.5 0.2.1 - 23.04.2016

Added:

1. add pyexcel-text file types as attributes of pyexcel.Sheet and pyexcel.Book, related to issue 31
2. auto import pyexcel-text if it is pip installed

Updated:

1. code refactored sources for easy addition of sources.
2. bug fix issue 29, Even if the format is a string it is displayed as a float
3. pyexcel-text is no longer a plugin to pyexcel-io but to pyexcel.sources, see pyexcel-text issue #22

Removed:

1. pyexcel.presentation is removed. No longer the internal decorate @outsource is used. related to issue 31

11.1.6 0.2.0 - 17.01.2016

Updated

1. adopt pyexcel-io yield key word to return generator as content
2. pyexcel.save_as and pyexcel.save_book_as get performance imporvements

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Symbols

__getitem__() (pyexcel.Sheet method), 72
__init__() (pyexcel.Book method), 59
__init__() (pyexcel.Sheet method), 63
__init__() (pyexcel.filters.ColumnFilter method), 88
__init__() (pyexcel.filters.ColumnValueFilter method), 90
__init__() (pyexcel.filters.EvenColumnFilter method), 90
__init__() (pyexcel.filters.EvenRowFilter method), 92
__init__() (pyexcel.filters.OddColumnFilter method), 89
__init__() (pyexcel.filters.OddRowFilter method), 91
__init__() (pyexcel.filters.RegionFilter method), 92
__init__() (pyexcel.filters.RowFilter method), 90
__init__() (pyexcel.filters.RowValueFilter method), 92
__init__() (pyexcel.filters.SingleColumnFilter method), 89
__init__() (pyexcel.filters.SingleRowFilter method), 91
__init__() (pyexcel.formatters.ColumnFormatter method), 86
__init__() (pyexcel.formatters.NamedColumnFormatter method), 87
__init__() (pyexcel.formatters.NamedRowFormatter method), 87
__init__() (pyexcel.formatters.RowFormatter method), 87
__init__() (pyexcel.formatters.SheetFormatter method), 88
__init__() (pyexcel.sheets.Column method), 102
__init__() (pyexcel.sheets.FilterableSheet method), 95
__init__() (pyexcel.sheets.FormattableSheet method), 94
__init__() (pyexcel.sheets.Matrix method), 93
__init__() (pyexcel.sheets.NamedColumn method), 85
__init__() (pyexcel.sheets.NamedRow method), 83
__init__() (pyexcel.sheets.NominableSheet method), 96
__init__() (pyexcel.sheets.Row method), 101
__init__() (pyexcel.sheets.Sheet method), 98

A

add_filter() (pyexcel.Sheet method), 78
add_formatter() (pyexcel.Sheet method), 76
apply_formatter() (pyexcel.Sheet method), 76

B

Book (class in pyexcel), 59

C

cell_value() (pyexcel.Sheet method), 72
clear_filters() (pyexcel.Sheet method), 78
clear_formatters() (pyexcel.Sheet method), 77
colnames (pyexcel.Sheet attribute), 75
Column (class in pyexcel.sheets), 102
column (pyexcel.Sheet attribute), 68
column_at() (pyexcel.Sheet method), 73
column_range() (pyexcel.Sheet method), 69
ColumnFilter (class in pyexcel.filters), 88
ColumnFormatter (class in pyexcel.formatters), 86
columns() (pyexcel.Sheet method), 70
ColumnValueFilter (class in pyexcel.filters), 90
cut() (pyexcel.Sheet method), 80

D

delete_columns() (pyexcel.Sheet method), 74
delete_named_column_at() (pyexcel.Sheet method), 75
delete_named_row_at() (pyexcel.Sheet method), 76
delete_rows() (pyexcel.Sheet method), 73
dict_to_array() (in module pyexcel), 79

E

enumerate() (pyexcel.Sheet method), 71
EvenColumnFilter (class in pyexcel.filters), 90
EvenRowFilter (class in pyexcel.filters), 92
extend_columns() (pyexcel.Sheet method), 74
extend_rows() (pyexcel.Sheet method), 73
extract_a_sheet_from_a_book() (in module pyexcel), 58

F

filter() (pyexcel.Sheet method), 78
FilterableSheet (class in pyexcel.sheets), 95
format() (pyexcel.Sheet method), 76
format() (pyexcel.sheets.NamedColumn method), 85
format() (pyexcel.sheets.NamedRow method), 84
FormattableSheet (class in pyexcel.sheets), 94

freeze_filters() (pyexcel.Sheet method), 78
freeze_formatters() (pyexcel.Sheet method), 77
from_records() (in module pyexcel), 79

G

get_array() (in module pyexcel), 54
get_book() (in module pyexcel), 55
get_book_dict() (in module pyexcel), 54
get_dict() (in module pyexcel), 54
get_records() (in module pyexcel), 54
get_sheet() (in module pyexcel), 55

M

map() (pyexcel.Sheet method), 79
Matrix (class in pyexcel.sheets), 93
merge_all_to_a_book() (in module pyexcel), 58
merge_csv_to_a_book() (in module pyexcel), 57

N

name_columns_by_row() (pyexcel.Sheet method), 74
name_rows_by_column() (pyexcel.Sheet method), 75
named_column_at() (pyexcel.Sheet method), 75
named_row_at() (pyexcel.Sheet method), 75
NamedColumn (class in pyexcel.sheets), 85
NamedColumnFormatter (class in pyexcel.formatters), 87
NamedRow (class in pyexcel.sheets), 83
NamedRowFormatter (class in pyexcel.formatters), 87
NominableSheet (class in pyexcel.sheets), 96
number_of_columns() (pyexcel.Sheet method), 69
number_of_rows() (pyexcel.Sheet method), 68
number_of_sheets() (pyexcel.Book method), 61

O

OddColumnFilter (class in pyexcel.filters), 89
OddRowFilter (class in pyexcel.filters), 91

P

paste() (pyexcel.Sheet method), 81

R

rcolumns() (pyexcel.Sheet method), 70
region() (pyexcel.Sheet method), 80
RegionFilter (class in pyexcel.filters), 92
remove_filter() (pyexcel.Sheet method), 78
remove_formatter() (pyexcel.Sheet method), 77
reverse() (pyexcel.Sheet method), 71
Row (class in pyexcel.sheets), 101
row (pyexcel.Sheet attribute), 68
row_at() (pyexcel.Sheet method), 72
row_range() (pyexcel.Sheet method), 69
RowFilter (class in pyexcel.filters), 90
RowFormatter (class in pyexcel.formatters), 87
rownames (pyexcel.Sheet attribute), 75
rows() (pyexcel.Sheet method), 69
RowValueFilter (class in pyexcel.filters), 92
rrows() (pyexcel.Sheet method), 70
rvertical() (pyexcel.Sheet method), 72

S

save_as() (in module pyexcel), 56
save_as() (pyexcel.Book method), 62
save_as() (pyexcel.Sheet method), 66
save_book_as() (in module pyexcel), 57
save_to() (pyexcel.Book method), 62
save_to() (pyexcel.Sheet method), 66
save_to_database() (pyexcel.Book method), 62
save_to_database() (pyexcel.Sheet method), 66
save_to_memory() (pyexcel.Book method), 62
save_to_memory() (pyexcel.Sheet method), 66
select() (pyexcel.sheets.NamedColumn method), 85
select() (pyexcel.sheets.NamedRow method), 84
set_column_at() (pyexcel.Sheet method), 74
set_named_column_at() (pyexcel.Sheet method), 75
set_named_row_at() (pyexcel.Sheet method), 75
set_row_at() (pyexcel.Sheet method), 73
Sheet (class in pyexcel), 63
Sheet (class in pyexcel.sheets), 98
sheet_names() (pyexcel.Book method), 61
SheetFormatter (class in pyexcel.formatters), 88
SingleColumnFilter (class in pyexcel.filters), 89
SingleRowFilter (class in pyexcel.filters), 91
split_a_book() (in module pyexcel), 58

T

to_array() (pyexcel.Sheet method), 78
to_dict() (pyexcel.Book method), 61
to_dict() (pyexcel.Sheet method), 79
to_records() (pyexcel.Sheet method), 79
transpose() (pyexcel.Sheet method), 79

V

vertical() (pyexcel.Sheet method), 71