pyexcel

Release 0.7.6

C.W.

Jul 01, 2026

CONTENTS

1 Introduction **3**

2 Support the project **5**

2.1 Installation . . . 5
2.2 Advanced usage :fire: . . . 6
2.3 Plugin shopping guide . . . 7
2.4 Design . . . 9
2.5 New tutorial . . . 16
2.6 Old tutorial . . . 47
2.7 Cook book . . . 79
2.8 Real world cases . . . 88
2.9 API documentation . . . 91
2.10 Developer's guide . . . 176
2.11 Change log . . . 181

3 Indices and tables **197**

Index **199**

Author
C.W.

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

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

License
New BSD License

Released
0.7.6

Generated
Jul 01, 2026

CHAPTER

ONE

INTRODUCTION

pyexcel provides **one** unified API for reading, manipulating, and writing data in various Excel formats. It simplifies the process of handling Excel files, making it an enjoyable task. Data in Excel files can be easily converted into *arrays or dictionaries* with minimal code, and vice versa. This library focuses **purely on data processing and does not address features like fonts, colors, or charts**.

The idea behind pyexcel originated from a common usability problem: when Excel-driven web applications are delivered to non-developer users (e.g., project assistants, human resources administrators), they often are not aware of the differences between file formats such as CSV, XLS, and XLSX. Rather than training users on these formats, pyexcel provides web developers with a unified interface to handle most Excel file types.

To add support for a specific Excel format in your application, simply install an additional pyexcel plugin—no code changes required. This eliminates issues with different file formats. In the broader community, pyexcel and its associated libraries aim to be a simple, easy-to-install alternative to Pandas, where minimal data manipulation is needed.

CHAPTER

TWO

SUPPORT THE PROJECT

If your company uses pyexcel and its components in a revenue-generating product, please consider supporting the project on GitHub or Patreon. Your financial support will enable me to dedicate more time to coding, improving documentation, and creating engaging content.

2.1 Installation

You can install pyexcel via pip:

```
$ pip install pyexcel
```

or clone it and install it:

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

you can easily save it into an excel file using the following code:

```
>>> import pyexcel
>>> # make sure you had pyexcel-xls installed
>>> a_list_of_dictionaries = [
...     {
...         "Name": 'Adam',
...         "Age": 28
...     },
...     {
...         "Name": 'Beatrice',
...         "Age": 29
...     },
...     {
...         "Name": 'Ceri',
...         "Age": 30
...     },
...     {
...         "Name": 'Dean',
...         "Age": 26
...     }
... ]
>>> pyexcel.save_as(records=a_list_of_dictionaries, dest_file_name="your_file.xls")
```

Here is what you will get in excel:

And here's how to read it back as records:

```
>>> import pyexcel as p
>>> records = p.iget_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
>>> p.free_resources()
```

Custom data rendering:

```
>>> # pip install pyexcel-text==0.2.7.1
>>> import pyexcel as p
>>> ccs_insight2 = p.Sheet()
>>> ccs_insight2.name = "Worldwide Mobile Phone Shipments (Billions), 2017-2021"
>>> ccs_insight2.ndjson = """
... {"year": ["2017", "2018", "2019", "2020", "2021"]}
... {"smart phones": [1.53, 1.64, 1.74, 1.82, 1.90]}
... {"feature phones": [0.46, 0.38, 0.30, 0.23, 0.17]}
... """.strip()
>>> ccs_insight2
pyexcel sheet:
+----------------+------+------+------+------+------+
| year           | 2017 | 2018 | 2019 | 2020 | 2021 |
+----------------+------+------+------+------+------+
| smart phones   | 1.53 | 1.64 | 1.74 | 1.82 | 1.9  |
+----------------+------+------+------+------+------+
| feature phones | 0.46 | 0.38 | 0.3  | 0.23 | 0.17 |
+----------------+------+------+------+------+------+
```

2.2 Advanced usage :fire:

If you are dealing with big data, please consider these usages:

```
>>> def increase_everyones_age(generator):
...     for row in generator:
...         row['Age'] += 1
...         yield row
>>> def duplicate_each_record(generator):
...     for row in generator:
...         yield row
...         yield row
>>> records = p.iget_records(file_name="your_file.xls")
>>> io=p.isave_as(records=duplicate_each_record(increase_everyones_age(records)),
...     dest_file_type='csv', dest_lineterminator='\n')
>>> print(io.getvalue())
Age,Name
29,Adam
29,Adam
30,Beatrice
```

(continues on next page)

(continued from previous page)

```
30,Beatrice
31,Ceri
31,Ceri
27,Dean
27,Dean
```

Two advantages of above method:

1. Add as many wrapping functions as you want.
2. Constant memory consumption

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

Table 1: A list of file formats supported by external plugins

Package name	Supported file formats	Dependencies
pyexcel-io	csv, csvz[1], tsv, tsvz[2]	csvz,tsvz readers depends on *chardet*
pyexcel-xls	xls, xlsx(read only), xlsm(read only)	xlrd, xlwt
pyexcel-xlsx	xlsx	openpyxl
pyexcel-ods3	ods	pyexcel-ezodf, lxml
pyexcel-ods	ods	odfpy

Table 2: Dedicated file reader and writers

Package name	Supported file formats	Dependencies
pyexcel-xlsxw	xlsx(write only)	XlsxWriter
pyexcel-libxlsxw	xlsx(write only)	libxlsxwriter
pyexcel-xlsxr	xlsx(read only)	lxml
pyexcel-xlsbr	xlsb(read only)	pyxlsb
pyexcel-odsr	read only for ods, fods	lxml
pyexcel-odsw	write only for ods	loxun
pyexcel-htmlr	html(read only)	lxml,html5lib
pyexcel-pdfr	pdf(read only)	camelot

2.3 Plugin shopping guide

Since 2020, all pyexcel-io plugins have dropped the support for python versions which are lower than 3.6. If you want to use any of those Python versions, please use pyexcel-io and its plugins versions that are lower than 0.6.0.

Except csv files, xls, xlsx and ods files are a zip of a folder containing a lot of xml files

The dedicated readers for excel files can stream read

In order to manage the list of plugins installed, you need to use pip to add or remove a plugin. When you use virtualenv, you can have different plugins per virtual environment. In the situation where you have multiple plugins that does the same thing in your environment, you need to tell pyexcel which plugin to use per function call. For example, pyexcel-ods and pyexcel-odsr, and you want to get_array to use pyexcel-odsr. You need to append get_array(..., library='pyexcel-odsr').

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

Table 3: Other data renderers

Package name	Supported file formats	Dependencies	Python versions
pyexcel-text	write only:rst, mediawiki, html, latex, grid, pipe, orgtbl, plain simple read only: ndjson r/w: json	tabulate	2.6, 2.7, 3.3, 3.4 3.5, 3.6, pypy
pyexcel-handsontable	handsontable in html	handsontable	same as above
pyexcel-pygal	svg chart	pygal	2.7, 3.3, 3.4, 3.5 3.6, pypy
pyexcel-sortable	sortable table in html	csvtotable	same as above
pyexcel-gantt	gantt chart in html	frappe-gantt	except pypy, same as above

For compatibility tables of pyexcel-io plugins, please click here

Table 4: Plugin compatibility table

pyexcel	pyexcel-io	pyexcel-text	pyexcel-handsontable	pyexcel-pygal	pyexcel-gantt
0.7.*+	0.6.2+	0.2.6+	0.0.1+	0.0.1	0.0.1
0.6.5+	0.6.2+	0.2.6+	0.0.1+	0.0.1	0.0.1
0.5.15+	0.5.19+	0.2.6+	0.0.1+	0.0.1	0.0.1
0.5.14	0.5.18	0.2.6+	0.0.1+	0.0.1	0.0.1
0.5.10+	0.5.11+	0.2.6+	0.0.1+	0.0.1	0.0.1
0.5.9.1+	0.5.9.1+	0.2.6+	0.0.1	0.0.1	0.0.1
0.5.4+	0.5.1+	0.2.6+	0.0.1	0.0.1	0.0.1
0.5.0+	0.4.0+	0.2.6+	0.0.1	0.0.1	0.0.1
0.4.0+	0.3.0+	0.2.5			

Table 5: A list of supported file formats

file format	definition
csv	comma separated values
tsv	tab separated values
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
xlsx	MS-Excel Extensions to the Office Open XML SpreadsheetML File Format.
xlsm	an MS-Excel Macro-Enabled Workbook file
ods	open document spreadsheet
fods	flat open document spreadsheet
json	JavaScript object notation
html	html table of the data structure
simple	simple presentation
rst	rStructured Text presentation of the data
mediawiki	media wiki table

2.4 Design

2.4.1 Introduction

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

Data models and data structures

When dealing with excel files, **pyexcel** pay attention to 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 formula, styling properties, this library ignores them and only pays 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*

Data source

A data source is a storage format of structured data. The most popular data source is an excel file. Libre Office/Microsoft Excel can easily be used to generate an excel file of your desired format. Besides a physical file, this library recognizes three additional types of source:

1. Excel files in computer memory. For example: when a file is uploaded to a Python server for information processing. If it is relatively small, it can 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 asks it to be sent to him.
3. Python structures. For example: a developer may have scraped a site and have stored data in Python array or dictionary. He may want to save this information as a file.

Reading from - and writing to - a data source is modelled as parsers and renderers in pyexcel. Excel data sources and database sources support read and write. Other data sources may only support read only, or write only methods.

Here is a list of data sources:

Data source	Read and write properties
Array	Read and write
Dictionary	Same as above
Records	Same as above
Excel files	Same as above
Excel files in memory	Same as above
Excel files on the web	Read only
Django models	Read and write
SQL models	Read and write
Database querysets	Read only
Textual sources	Write only

Data format

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

file format	definition
csv	comma separated values
tsv	tab separated values
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
html	html table of the data structure
simple	*simple* presentation
rst	rStructured Text presentation of the data
mediawiki	media wiki table

See also *A list of file formats supported by external plugins*.

Data transformation

Often a developer would like to have excel data imported into a Python data structure. This library supports the *conversions from* previous three data source to the following list of data structures, and *vice versa*.

Table 6: A list of supported data structures

Pesudo name	Python name	Related model
two dimensional array	a list of lists	*pyexcel.Sheet*
a dictionary of key value pair	a dictionary	*pyexcel.Sheet*
a dictionary of one dimensional arrays	a dictionary of lists	*pyexcel.Sheet*
a list of dictionaries	a list of dictionaries	*pyexcel.Sheet*
a dictionary of two dimensional arrays	a dictionary of lists of lists	*pyexcel.Book*

Data manipulation

The main operation on a cell involves *cell access*, *formatting* and *cleansing*. The main operation on a sheet involves 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.

Data transcoding

For various reasons the data in one format needs to be transcoded into another. This library provides a transcoding tunnel for data transcoding between supported file formats.

Data visualization

Via `pyexcel.renderer.AbstractRenderer` interface, data visualization is made possible. **pyexcel-chart** is the interface plugin to formalize this effort. **pyexcel-pygal** is the first plugin to provide bar, pie, histogram charts and more.

[1] quoted from whatis.com. Technical details can be found at MSDN XLS
[2] xlsx is used by MS-Excel 2007, more information can be found at MSDN XLSX

Examples of supported data structure

Here is a list of examples:

```
>>> import pyexcel as p
>>> two_dimensional_list = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
>>> p.get_sheet(array=two_dimensional_list)
pyexcel_sheet1:
+---+----+----+----+
| 1 | 2  | 3  | 4  |
+---+----+----+----+
| 5 | 6  | 7  | 8  |
+---+----+----+----+
| 9 | 10 | 11 | 12 |
+---+----+----+----+
>>> a_dictionary_of_key_value_pair = {
...     "IE": 0.2,
...     "Firefox": 0.3
... }
>>> p.get_sheet(adict=a_dictionary_of_key_value_pair)
pyexcel_sheet1:
+---------+-----+
| Firefox | IE  |
+---------+-----+
| 0.3     | 0.2 |
+---------+-----+
>>> a_dictionary_of_one_dimensional_arrays = {
...      "Column 1": [1, 2, 3, 4],
...      "Column 2": [5, 6, 7, 8],
...      "Column 3": [9, 10, 11, 12],
... }
>>> p.get_sheet(adict=a_dictionary_of_one_dimensional_arrays)
pyexcel_sheet1:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+----------+----------+----------+
| 1        | 5        | 9        |
+----------+----------+----------+
| 2        | 6        | 10       |
+----------+----------+----------+
| 3        | 7        | 11       |
+----------+----------+----------+
| 4        | 8        | 12       |
+----------+----------+----------+
>>> a_list_of_dictionaries = [
...     {
...         "Name": 'Adam',
...         "Age": 28
...     },
...     {
```

(continues on next page)

(continued from previous page)

```
...         "Name": 'Beatrice',
...         "Age": 29
...     },
...     {
...         "Name": 'Ceri',
...         "Age": 30
...     },
...     {
...         "Name": 'Dean',
...         "Age": 26
...     }
... ]
>>> p.get_sheet(records=a_list_of_dictionaries)
pyexcel_sheet1:
+-----+----------+
| Age | Name     |
+-----+----------+
| 28  | Adam     |
+-----+----------+
| 29  | Beatrice |
+-----+----------+
| 30  | Ceri     |
+-----+----------+
| 26  | Dean     |
+-----+----------+
>>> 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]
...          ]
...  }
>>> p.get_book(bookdict=a_dictionary_of_two_dimensional_arrays)
Sheet 1:
+-----+-----+-----+
| 1.0 | 2.0 | 3.0 |
+-----+-----+-----+
| 4.0 | 5.0 | 6.0 |
+-----+-----+-----+
```

(continues on next page)

(continued from previous page)

```
| 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 |
+-----+-----+-----+
```

2.4.2 Signature functions

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 structures: dictionary, records and array

and to transform them into one of the following data structures:

- two dimensional 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

Python data can be handled well using lists, dictionaries and various mixture of both. This library provides four module level functions to help you obtain excel data in these data structures. 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 7: A list of module level functions

Functions	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:

- 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*

The following two variants of the data access function use generator and should work well with big data files

Table 8: A list of variant functions

Functions	Name	Python name
iget	**a memory efficient two dimensional** array	a generator of a list of lists
iget	a memory efficient list list of dictionaries	a generator of a list of dictionaries

However, you will need to call `free_resource()` to make sure file handles are closed.

Two pyexcel 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.

Export data from Python

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

- two dimensional 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 structures: dictionary, records and array

Here are the two functions:

Functions	Description
save	Works well with single sheet file
isav	Works well with big data files
save	**Works with multiple sheet file** and big data files
isav	**Works with multiple sheet file** and big data files

If you would only use these two functions to do format transcoding, you may enjoy a speed boost using *isave_as()* and *isave_book_as()*, because they use *yield* keyword and minimize memory footprint. However, you will need to call `free_resource()` to make sure file handles are closed. And *save_as()* and *save_book_as()* reads all data into memory and **will make all rows the same width**.

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*

Data transportation/transcoding

This library is capable of transporting your data between any of the following data sources:

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

See also:

- *How to import 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*

2.4.3 Architecture

pyexcel uses loosely couple plugins to fullfil the promise to access various file formats. **lml** is the plugin management library that provide the specialized support for the loose coupling.

What is loose coupling?

The components of **pyexcel** is designed as building blocks. For your project, you can cherry-pick the file format support without affecting the core functionality of pyexcel. Each plugin will bring in additional dependences. For example, if you choose pyexcel-xls, xlrd and xlwt will be brought in as 2nd level depndencies.

Looking at the following architectural diagram, pyexcel hosts plugin interfaces for data source, data renderer and data parser. pyexcel-pygal, pyexcel-matplotlib, and pyexcel-handsontable extend pyexcel using data renderer interface. pyexcel-io package takes away the responsibilities to interface with excel libraries, for example: xlrd, openpyxl, ezodf.

As in *A list of file formats supported by external plugins*, there are overlapping capabilities in reading and writing xlsx, ods files. Because each third parties express different personalities although they may read and write data in the same file format, you as the pyexcel is left to pick which suit your task best.

Dotted arrow means the package or module is loaded later.

2.5 New tutorial

2.5.1 One liners

This section shows you how to get data from your excel files and how to export data to excel files in **one line**

Read from the excel files

Get a list of dictionaries

Suppose you want to process History of Classical Music:

Let's get a list of dictionary out from the xls file:

```
>>> records = p.get_records(file_name="your_file.xls")
```

And let's check what do we have:

```
>>> for row in records:
...     print(f"{row['Representative Composers']} are from {row['Name']} period ({row[
↪'Period']})")
Machaut, Landini are from Medieval period (c.1150-c.1400)
Gibbons, Frescobaldi are from Renaissance period (c.1400-c.1600)
JS Bach, Vivaldi are from Baroque period (c.1600-c.1750)
Joseph Haydn, Wolfgan Amadeus Mozart are from Classical period (c.1750-c.1830)
Chopin, Mendelssohn, Schumann, Liszt are from Early Romantic period (c.1830-c.1860)
Wagner,Verdi are from Late Romantic period (c.1860-c.1920)
Sergei Rachmaninoff,Calude Debussy are from Modernist period (20th century)
```

Get two dimensional array

Instead, what if you have to use *pyexcel.get_array* to do the same:

```
>>> for row in p.get_array(file_name="your_file.xls", start_row=1):
...     print(f"{row[2]} are from {row[0]} period ({row[1]})")
Machaut, Landini are from Medieval period (c.1150-c.1400)
Gibbons, Frescobaldi are from Renaissance period (c.1400-c.1600)
JS Bach, Vivaldi are from Baroque period (c.1600-c.1750)
Joseph Haydn, Wolfgan Amadeus Mozart are from Classical period (c.1750-c.1830)
Chopin, Mendelssohn, Schumann, Liszt are from Early Romantic period (c.1830-c.1860)
Wagner,Verdi are from Late Romantic period (c.1860-c.1920)
Sergei Rachmaninoff,Calude Debussy are from Modernist period (20th century)
```

where *start_row* skips the header row.

Get a dictionary

You can get a dictionary too:

```
>>> my_dict = p.get_dict(file_name="your_file.xls", name_columns_by_row=0)
```

And let's have a look inside:

```
>>> from pyexcel._compact import OrderedDict
>>> isinstance(my_dict, OrderedDict)
True
>>> for key, values in my_dict.items():
...     print(key + " : " + ','.join([str(item) for item in values]))
Name : Medieval,Renaissance,Baroque,Classical,Early Romantic,Late Romantic,Modernist
Period : c.1150-c.1400,c.1400-c.1600,c.1600-c.1750,c.1750-c.1830,c.1830-c.1860,c.1860-c.
↪1920,20th century
Representative Composers : Machaut, Landini,Gibbons, Frescobaldi,JS Bach, Vivaldi,Joseph␣
↪Haydn, Wolfgan Amadeus Mozart,Chopin, Mendelssohn, Schumann, Liszt,Wagner,Verdi,Sergei␣
↪Rachmaninoff,Calude Debussy
```

Please note that my_dict is an OrderedDict.

Get a dictionary of two dimensional array

Suppose you have a multiple sheet book as the following:

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

```
>>> book_dict = p.get_book_dict(file_name="book.xls")
```

And check:

```
>>> isinstance(book_dict, OrderedDict)
True
>>> import json
>>> for key, item in book_dict.items():
...     print(json.dumps({key: item}))
{"Most Expensive Violins": [["Name", "Estimated Value", "Location"], ["Messiah␣
↪Stradivarious", "$ 20,000,000", "Ashmolean Museum in Oxford, England"], ["Vieuxtemps␣
↪Guarneri", "$ 16,000,000", "On loan to Anne Akiko Meyers"], ["Lady Blunt", "$ 15,900,
↪000", "Anonymous bidder"]]}
{"Noteable Violin Makers": [["Maker", "Period", "Country"], ["Antonio Stradivari", "1644-
↪1737", "Cremona, Italy"], ["Giovanni Paolo Maggini", "1580-1630", "Botticino, Italy"],␣
↪["Amati Family", "1500-1740", "Cremona, Italy"], ["Guarneri Family", "1626-1744",
↪"Cremona, Italy"], ["Rugeri Family", "1628-1719", "Cremona, Italy"], ["Carlo Bergonzi",
↪ "1683-1747", "Cremona, Italy"], ["Jacob Stainer", "1617-1683", "Austria"]]}
{"Top Violinist": [["Name", "Period", "Nationality"], ["Antonio Vivaldi", "1678-1741",
↪"Italian"], ["Niccolo Paganini", "1782-1840", "Italian"], ["Pablo de Sarasate", "1852-
↪1904", "Spainish"], ["Eugene Ysaye", "1858-1931", "Belgian"], ["Fritz Kreisler", "1875-
↪1962", "Astria-American"], ["Jascha Heifetz", "1901-1987", "Russian-American"], [
↪"David Oistrakh", "1908-1974", "Russian"], ["Yehundi Menuhin", "1916-1999", "American
↪"], ["Itzhak Perlman", "1945-", "Israeli-American"], ["Hilary Hahn", "1979-", "American
↪"]]}
```

Write data

Export an array

Suppose you have the following array:

```
>>> data = [['G', 'D', 'A', 'E'], ['Thomastik-Infeld Dominants', 'Thomastik-Infeld␣
↪Dominants', 'Thomastik-Infeld Dominants', 'Pirastro'], ['Silver wound', '', 'Aluminum␣
↪wound', 'Gold Label Steel']]
```

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

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

Let's verify it:

```
>>> p.get_sheet(file_name="example.xls")
pyexcel_sheet1:
+----------------------------+----------------------------+----------------------------+-
↪-----------------+
| G                          | D                          | A                          |␣
↪E                 |
+----------------------------+----------------------------+----------------------------+-
```

(continues on next page)

(continued from previous page)

```
↪-----------------+
| Thomastik-Infeld Dominants | Thomastik-Infeld Dominants | Thomastik-Infeld Dominants |␣
↪Pirastro         |
+----------------------------+----------------------------+----------------------------+-
↪-----------------+
| Silver wound               |                            | Aluminum wound             |␣
↪Gold Label Steel |
+----------------------------+----------------------------+----------------------------+-
↪-----------------+
```

And here is the code to save it as a csv file :

```
>>> p.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())
...
G:D:A:E
Thomastik-Infeld Dominants:Thomastik-Infeld Dominants:Thomastik-Infeld Dominants:Pirastro
Silver wound::Aluminum wound:Gold Label Steel
```

Export a list of dictionaries

```
>>> records = [
...     {"year": 1903, "country": "Germany", "speed": "206.7km/h"},
...     {"year": 1964, "country": "Japan", "speed": "210km/h"},
...     {"year": 2008, "country": "China", "speed": "350km/h"}
... ]
>>> p.save_as(records=records, dest_file_name='high_speed_rail.xls')
```

Export a dictionary of single key value pair

```
>>> henley_on_thames_facts = {
...     "area": "5.58 square meters",
...     "population": "11,619",
...     "civil parish": "Henley-on-Thames",
...     "latitude": "51.536",
...     "longitude": "-0.898"
... }
>>> p.save_as(adict=henley_on_thames_facts, dest_file_name='henley.xlsx')
```

Export a dictionary of single dimensional array

```
>>> ccs_insights = {
...     "year": ["2017", "2018", "2019", "2020", "2021"],
```

(continues on next page)

(continued from previous page)

```
...     "smart phones": [1.53, 1.64, 1.74, 1.82, 1.90],
...     "feature phones": [0.46, 0.38, 0.30, 0.23, 0.17]
... }
>>> p.save_as(adict=ccs_insights, dest_file_name='ccs.csv')
```

Export a dictionary of two dimensional array as a book

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

```
>>> a_dictionary_of_two_dimensional_arrays = {
...     'Top 3 Aircraft Manufacturers':
...         [
...             ['Name', 'Revenue'],
...             ['Lockheed Martin', '65.4 billion USD'],
...             ['Airbus', '78.9 billion USD'],
...             ['Boeing', '58.16 billion USD']
...         ],
...     'Top 3 Airlines':
...         [
...             ['Name', 'Country', 'Revenue'],
...             ['Delta Air Lines', 'US', 61.6],
...             ['American Airlines Holdings', 'US', 57.1],
...             ['American Airlines Group', 'US', 54.2]
...         ],
...     'Biggest 3 Airoplanes':
...         [
...             ['Model', 'Passenger limt'],
...             ['Airbus A380-800', 853],
...             ['Boeing 747-400', 660],
...             ['Boeing 747-8', 605]
...         ]
...  }
```

Here is the code:

```
>>> p.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['Top 3 Airlines']})
>>> data.update({"Sheet 1": a_dictionary_of_two_dimensional_arrays['Top 3 Aircraft␣
↪Manufacturers']})
>>> data.update({"Sheet 3": a_dictionary_of_two_dimensional_arrays['Biggest 3 Airoplanes
↪']})
>>> p.save_book_as(bookdict=data, dest_file_name="book.xls")
```

Let's verify its order:

```
>>> book_dict = p.get_book_dict(file_name="book.xls")
>>> for key, item in book_dict.items():
...     print(json.dumps({key: item}))
{"Sheet 2": [["Name", "Country", "Revenue"], ["Delta Air Lines", "US", 61.6], ["American␣
↪Airlines Holdings", "US", 57.1], ["American Airlines Group", "US", 54.2]]}
{"Sheet 1": [["Name", "Revenue"], ["Lockheed Martin", "65.4 billion USD"], ["Airbus",
↪"78.9 billion USD"], ["Boeing", "58.16 billion USD"]]}
{"Sheet 3": [["Model", "Passenger limt"], ["Airbus A380-800", 853], ["Boeing 747-400",␣
↪660], ["Boeing 747-8", 605]]}
```

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

Transcoding

Note

Please note that *pyexcel-cli* can perform file transcoding at command line. No need to open your editor, save the code, then python run.

The following code does a simple file format transcoding from xls to csv:

```
>>> p.save_as(file_name="trump_tariffs.xls", dest_file_name="trump_tariffs.csv")
```

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

```
>>> sheet = p.get_sheet(file_name="trump_tariffs.csv")
>>> sheet
trump_tariffs.csv:
+----------------+--------------------+-----------------------------+
| Country        | New US tariffs, % | Tariffs charged to the USA |
+----------------+--------------------+-----------------------------+
| China          | 34                 | 67                          |
+----------------+--------------------+-----------------------------+
| EU             | 20                 | 39                          |
+----------------+--------------------+-----------------------------+
| United Kingdom | 10                 | 10                          |
+----------------+--------------------+-----------------------------+
```

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.

Let use previous example and save it as xlsx instead

```
>>> p.save_as(file_name="trump_tariffs.xls",
...           dest_file_name="trump_tariffs.xlsx") # change the file extension
```

Again let's verify what we have gotten:

```
>>> sheet = p.get_sheet(file_name="trump_tariffs.xlsx")
>>> sheet
pyexcel_sheet1:
+----------------+--------------------+-----------------------------+
| Country        | New US tariffs, % | Tariffs charged to the USA |
+----------------+--------------------+-----------------------------+
| China          | 34                 | 67                          |
+----------------+--------------------+-----------------------------+
| EU             | 20                 | 39                          |
+----------------+--------------------+-----------------------------+
| United Kingdom | 10                 | 10                          |
+----------------+--------------------+-----------------------------+
```

Excel book merge and split operation in one line

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")
```

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

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

2.5.2 Stream APIs for big file : A set of two liners

When you are dealing with **BIG** excel files, you will want **pyexcel** to use constant memory.

This section shows you how to get data from your **BIG** excel files and how to export data to excel files in **two lines** at most, without eating all your computer memory.

Two liners for get data from big excel files

Get a list of dictionaries

Suppose you want to process the following coffee data:

Let's get a list of dictionary out from the xls file:

```
>>> records = p.iget_records(file_name="your_file.xls")
```

And let's check what do we have:

```
>>> for r in records:
...     print(f"{r['Serving Size']} of {r['Coffees']} has {r['Caffeine (mg)']} mg")
venti(20 oz) of Starbucks Coffee Blonde Roast has 475 mg
large(20 oz.) of Dunkin' Donuts Coffee with Turbo Shot has 398 mg
grande(16 oz.) of Starbucks Coffee Pike Place Roast has 310 mg
regular(16 oz.) of Panera Coffee Light Roast has 300 mg
```

Please do not forget the second line to close the opened file handle:

```
>>> p.free_resources()
```

Get two dimensional array

Instead, what if you have to use *pyexcel.get_array* to do the same:

```
>>> for row in p.iget_array(file_name="your_file.xls", start_row=1):
...     print(f"{row[1]} of {row[0]} has {row[2]} mg")
venti(20 oz) of Starbucks Coffee Blonde Roast has 475 mg
large(20 oz.) of Dunkin' Donuts Coffee with Turbo Shot has 398 mg
grande(16 oz.) of Starbucks Coffee Pike Place Roast has 310 mg
regular(16 oz.) of Panera Coffee Light Roast has 300 mg
```

Again, do not forget the second line:

```
>>> p.free_resources()
```

where *start_row* skips the header row.

Data export in one liners

Export an array

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 :

```
>>> p.isave_as(array=data, dest_file_name="example.xls")
```

But the following line is not required because the data source are not file sources:

```
>>> # p.free_resources()
```

Let's verify it:

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

And here is the code to save it as a csv file :

```
>>> p.isave_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
```

Export a list of dictionaries

```
>>> records = [
...     {"year": 1903, "country": "Germany", "speed": "206.7km/h"},
...     {"year": 1964, "country": "Japan", "speed": "210km/h"},
```

(continues on next page)

(continued from previous page)

```
...     {"year": 2008, "country": "China", "speed": "350km/h"},
...     {"year": 2025, "country": "China", "speed": "400km/h"}
... ]
>>> p.isave_as(records=records, dest_file_name='high_speed_rail.xls')
```

Export a dictionary of single key value pair

```
>>> henley_on_thames_facts = {
...     "area": "5.58 square meters",
...     "population": "11,619",
...     "civil parish": "Henley-on-Thames",
...     "latitude": "51.536",
...     "longitude": "-0.898"
... }
>>> p.isave_as(adict=henley_on_thames_facts, dest_file_name='henley.xlsx')
```

Export a dictionary of single dimensional array

```
>>> ccs_insights = {
...     "year": ["2017", "2018", "2019", "2020", "2021"],
...     "smart phones": [1.53, 1.64, 1.74, 1.82, 1.90],
...     "feature phones": [0.46, 0.38, 0.30, 0.23, 0.17]
... }
>>> p.isave_as(adict=ccs_insights, dest_file_name='ccs.csv')
>>> p.free_resources()
```

Export a dictionary of two dimensional array as a book

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:

```
>>> p.isave_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:

```
>>> from pyexcel._compact import OrderedDict
>>> 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']})
>>> p.isave_book_as(bookdict=data, dest_file_name="book.xls")
>>> p.free_resources()
```

Let's verify its order:

```
>>> import json
>>> book_dict = p.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.

File format transcoding on one line

> **Note**
>
> Please note that the following file transcoding could be with zero line. Please install pyexcel-cli and you will do the transcode in one command. No need to open your editor, save the problem, then python run.

The following code does a simple file format transcoding from xls to csv:

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

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

```
>>> sheet = p.get_sheet(file_name="birth.csv")
>>> sheet
birth.csv:
+-------+--------+----------+
| name  | weight | birth    |
+-------+--------+----------+
| Adam  | 3.4    | 03/02/15 |
+-------+--------+----------+
```

(continues on next page)

(continued from previous page)

```
| 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.

Let use previous example and save it as xlsx instead

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

Again let's verify what we have gotten:

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

2.5.3 For web developer

The following libraries are written to facilitate the daily import and export of excel data.

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

And you may make your own by using pyexcel-webio

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

```
def upload():
    if request.method == 'POST' and 'excel' in request.files:
        # handle file upload
        filename = request.files['excel'].filename
        extension = filename.split(".")[-1]
        # Obtain the file extension and content
        # pass a tuple instead of a file name
```

(continues on next page)

(continued from previous page)

```
        content = request.files['excel'].read()
        if sys.version_info[0] > 2:
            # in order to support python 3
            # have to decode bytes to str
            content = content.decode('utf-8')
        sheet = pe.get_sheet(file_type=extension, file_content=content)
        # then use it as usual
        sheet.name_columns_by_row(0)
        # respond with a json
        return jsonify({"result": sheet.dict})
    return render_template('upload.html')
```

request.files['excel'] in line 4 holds the file object. line 5 finds out the file extension. line 13 obtains a sheet instance. line 15 uses the first row as data header. line 17 sends the json representation of the excel file back to client browser.

Write to memory and respond to download

```
data = [
    [...],
    ...
]

@app.route('/download')
def download():
    sheet = pe.Sheet(data)
    output = make_response(sheet.csv)
    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

2.5.4 Pyexcel data renderers

There exist a few data renderers for pyexcel data. This chapter will walk you through them.

View pyexcel data in ndjson and other formats

With pyexcel-text, you can get pyexcel data in newline delimited json, normal json and other formats.

View the pyexcel data in a browser

You can use pyexcel-handsontable to render your data.

Include excel data in your python documentation

sphinxcontrib-excel help you present your excel data in various formats inside your sphinx documentation.

Draw charts from your excel data

pyexcel-pygal helps you with all charting options and give you charts in svg format.

pyexcel-echarts draws 2D, 3D, geo charts from pyexcel data and has awesome animations too, but it is under development.

pyexcel-matplotlib helps you with scientific charts and is under development.

Gantt chart visualization for your excel data

pyexcel-gantt is a specialist renderer for gantt chart.

2.5.5 Sheet

The sheet api here is much less powerful than pandas DataFrame when the array is of significant size. To be honesty, pandas DataFrame is much more powerful and provide rich data manipulation apis. When would you consider the sheet api here? if your data manipulation steps are basic and your data volume is not high, you can use them.

Random access

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].

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

```
>>> 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
>>> sheet[3, 3] = 10
>>> print(sheet[3, 3])
10
```

> **Note**
>
> In order to set a value to a cell, please use sheet[row_index, column_index] = new_value
>
> or sheet['A1'] = new_value

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]
```

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
>>> sheet[1, "Y"] = 100
>>> print(sheet[1, "Y"])
100
```

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, 100, 8]
```

Hence access the same cell, this statement also works:

```
>>> sheet.column['Y'][1]
100
```

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]
100
```

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"])
100
>>> sheet["b", "Y"] = 200
>>> print(sheet["b", "Y"])
200
```

Play with data

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

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

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:

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

attributes

Attributes:

```
>>> import pyexcel
>>> content = "1,2,3\n3,4,5"
>>> sheet = pyexcel.get_sheet(file_type="csv", file_content=content)
>>> sheet.tsv
'1\t2\t3\r\n3\t4\t5\r\n'
>>> print(sheet.simple)
csv:
-  -  -
1  2  3
3  4  5
-  -  -
```

What's more, you could as well set value to an attribute, for example::

```
>>> import pyexcel
>>> content = "1,2,3\n3,4,5"
>>> sheet = pyexcel.Sheet()
>>> sheet.csv = content
>>> sheet.array
[[1, 2, 3], [3, 4, 5]]
```

You can get the direct access to underneath stream object. In some situation, it is desired:

```
>>> stream = sheet.stream.tsv
```

The returned stream object has tsv formatted content for reading.

What you could further do is to set a memory stream of any supported file format to a sheet. For example:

```
>>> another_sheet = pyexcel.Sheet()
>>> another_sheet.xls = sheet.xls
>>> another_sheet.content
+---+---+---+
```

(continues on next page)

(continued from previous page)

```
| 1 | 2 | 3 |
+---+---+---+
| 3 | 4 | 5 |
+---+---+---+
```

Yet, it is possible assign a absolute url to an online excel file to an instance of *pyexcel.Sheet*.

custom attributes

You can pass on source specific parameters to getter and setter functions.

```
>>> content = "1-2-3\n3-4-5"
>>> sheet = pyexcel.Sheet()
>>> sheet.set_csv(content, delimiter="-")
>>> sheet.csv
'1,2,3\r\n3,4,5\r\n'
>>> sheet.get_csv(delimiter="|")
'1|2|3\r\n3|4|5\r\n'
```

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 exposed by `Column` and `Row`

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        |
```

(continues on next page)

(continued from previous page)

```
+----------+----------+----------+
| 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        |
+----------+----------+
```

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)
>>> sheet3 = sheet.column + sheet2
>>> sheet3.column["Column 4"]
[10, 11, 12]
>>> sheet3.column["Column 5"]
[13, 14, 15]
```

Please note above column plus statement will not update original *sheet* instance, as pyexcel user demanded:

```
>>> sheet
pyexcel sheet:
+----------+----------+
| Column 1 | Column 3 |
+==========+==========+
| 1        | 7        |
+----------+----------+
| 2        | 8        |
+----------+----------+
| 3        | 9        |
+----------+----------+
```

So, to change orginal *sheet* instance, you can elect to do:

```
>>> sheet.column += sheet2
```

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:
```

(continues on next page)

(continued from previous page)

```
+---+---+---+---+
| b | d | f | g |
+===+===+===+===+
| 2 | 4 | 6 | 7 |
+---+---+---+---+
```

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        |
+----------+----------+----------+
```

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)
>>> sheet
pyexcel sheet:
+-------+---+---+---+
| Row 1 | a | b | c |
+-------+---+---+---+
| Row 2 | e | f | g |
+-------+---+---+---+
| Row 3 | 1 | 2 | 3 |
+-------+---+---+---+
```

Then you can access rows by its name:

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

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. use *format()* to apply formatter immediately.

Note

int, **float** and **datetime** values are automatically detected in **csv** files since **pyexcel** version 0.2.2

Convert a column of numbers to strings

Suppose you have the following data:

```
>>> import pyexcel
>>> data = [
...     ["userid","name"],
...     [10120,"Adam"],
...     [10121,"Bella"],
...     [10122,"Cedar"]
... ]
>>> sheet = pyexcel.Sheet(data)
>>> sheet.name_columns_by_row(0)
>>> sheet.column["userid"]
[10120, 10121, 10122]
```

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

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

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 " ":

```
>>> data = [
...     ["        Version", "        Comments", "       Author  "],
...     ["  v0.0.1       ", " Release versions","            Eda"],
...     ["  v0.0.2  ", "Useful updates    ", "   Freud"]
... ]
>>> sheet = pyexcel.Sheet(data)
>>> sheet.content
+-----------------+------------------------------+----------------------+
|         Version |         Comments             |        Author   |
+-----------------+------------------------------+----------------------+
|   v0.0.1        |  Release versions            |             Eda |
+-----------------+------------------------------+----------------------+
|   v0.0.2   | Useful updates     |    Freud        |
+-----------------+------------------------------+----------------------+
```

Now try to create a custom cleanse function:

```
.. code-block:: python
```

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

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

```
.. code-block:: python
```

```
>>> sheet.map(cleanse_func)
```

So in the end, you get this:

```
>>> sheet.content
+---------+------------------+--------+
| Version | Comments         | Author |
+---------+------------------+--------+
| v0.0.1  | Release versions | Eda    |
+---------+------------------+--------+
| v0.0.2  | Useful updates   | Freud  |
+---------+------------------+--------+
```

Data 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)
>>> sheet.content
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+==========+==========+==========+
| 1        | 2        | 3        |
+----------+----------+----------+
| 4        | 5        | 6        |
+----------+----------+----------+
| 7        | 8        | 9        |
+----------+----------+----------+
```

Filter out some data

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

```
>>> sheet.filter(row_indices=[0, 2])
>>> sheet.content
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+==========+==========+==========+
| 4        | 5        | 6        |
+----------+----------+----------+
```

Let's try to further filter out even columns:

```
>>> sheet.filter(column_indices=[1])
>>> sheet.content
+----------+----------+
| Column 1 | Column 3 |
+==========+==========+
| 4        | 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

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]])
```

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_index, row):
...     result = [element for element in row if element != '']
...     return len(result)==0
```

And then apply the filter on the sheet:

```
>>> del sheet.row[filter_row]
>>> sheet
pyexcel sheet:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
```

2.5.6 Book

You access each cell via this syntax:

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

or:

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

Suppose you have the following sheets:

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 as an two dimensional array and multi-sheet excel book as an ordered dictionary of two dimensional arrays.

Write multiple sheet excel book

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

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
```

Get content

```
>>> book_dict = {
...      '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]
...          ],
...      'Sheet 1':
```

(continues on next page)

(continued from previous page)

```
...           [
...               [1.0, 2.0, 3.0],
...               [4.0, 5.0, 6.0],
...               [7.0, 8.0, 9.0]
...           ]
...   }
>>> book = pyexcel.get_book(bookdict=book_dict)
>>> book
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 |
+-----+-----+-----+
>>> print(book.rst)
Sheet 1:
=  =  =
1  2  3
4  5  6
7  8  9
=  =  =
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
===  ===  ===
```

You can get the direct access to underneath stream object. In some situation, it is desired.

```
>>> stream = book.stream.plain
```

The returned stream object has the content formatted in plain format for further reading.

Set content

Surely, you could set content to an instance of *pyexcel.Book*.

```
>>> other_book = pyexcel.Book()
>>> other_book.bookdict = book_dict
>>> print(other_book.plain)
Sheet 1:
1  2  3
4  5  6
7  8  9
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
```

You can set via 'xls' attribute too.

```
>>> another_book = pyexcel.Book()
>>> another_book.xls = other_book.xls
>>> print(another_book.mediawiki)
Sheet 1:
{| class="wikitable" style="text-align: left;"
|+ <!-- caption -->
|-
| align="right"| 1 || align="right"| 2 || align="right"| 3
|-
| align="right"| 4 || align="right"| 5 || align="right"| 6
|-
| align="right"| 7 || align="right"| 8 || align="right"| 9
|}
Sheet 2:
{| class="wikitable" style="text-align: left;"
|+ <!-- caption -->
|-
| X || Y || Z
|-
| 1 || 2 || 3
|-
| 4 || 5 || 6
|}
Sheet 3:
{| class="wikitable" style="text-align: left;"
|+ <!-- caption -->
|-
```

(continues on next page)

(continued from previous page)

```
| O || P || Q
|-
| 3 || 2 || 1
|-
| 4 || 3 || 2
|}
```

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 |
+---+---+---+
```

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 individual 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"]
```

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

How do I read a book, process 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)
```

(continues on next page)

(continued from previous page)

```
...
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

Alternatively, you could use *save_book_as()* function

```
>>> pyexcel.save_book_as(bookdict=content, dest_file_name="myfile.csv")
```

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

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:
+-----+-----+-----+
| 1.0 | 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 |
+-----+-----+-----+
```

2.5.7 Working with databases

How to import 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:

```
>>> import os
>>> import pyexcel as p
>>> 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:

Here is the code to import it:

```
>>> session = Session() # obtain a sql session
>>> p.save_as(file_name="birth.xls", name_columns_by_row=0, dest_session=session, dest_
↪table=BirthRegister)
```

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

```
>>> sheet = p.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    |
+------------+----+-------+--------+
```

2.6 Old tutorial

2.6.1 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 styles.

Open a csv file

Read a csv file is simple:

```
>>> import pyexcel as p
>>> sheet = p.get_sheet(file_name="example.csv")
>>> sheet
example.csv:
+---+---+---+
| 1 | 4 | 7 |
+---+---+---+
| 2 | 5 | 8 |
+---+---+---+
| 3 | 6 | 9 |
+---+---+---+
```

The same applies to a tsv file:

```
>>> sheet = p.get_sheet(file_name="example.tsv")
>>> sheet
example.tsv:
+---+---+---+
| 1 | 4 | 7 |
+---+---+---+
| 2 | 5 | 8 |
+---+---+---+
| 3 | 6 | 9 |
+---+---+---+
```

Meanwhile, a tab separated file can be read as csv too. You can specify a delimiter parameter.

```
>>> with open('tab_example.csv', 'w') as f:
...     unused = f.write('I\tam\ttab\tseparated\tcsv\n') # for passing doctest
...     unused = f.write('You\tneed\tdelimiter\tparameter\n') # unused is added
>>> sheet = p.get_sheet(file_name="tab_example.csv", delimiter='\t')
>>> sheet
tab_example.csv:
+-----+------+-----------+-----------+-----+
| I   | am   | tab       | separated | csv |
+-----+------+-----------+-----------+-----+
| You | need | delimiter | parameter |     |
+-----+------+-----------+-----------+-----+
```

Add a new row to an existing file

Suppose you have one data file as the following:

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_sheet1:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+----------+----------+----------+
| 1        | 4        | 7        |
+----------+----------+----------+
| 2        | 5        | 8        |
+----------+----------+----------+
| 3        | 6        | 9        |
+----------+----------+----------+
| 12       | 11       | 10       |
+----------+----------+----------+
```

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:

```
.. code-block:: python
```

```
>>> 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_sheet1:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+----------+----------+----------+
| 1        | 4        | 7        |
+----------+----------+----------+
| 2        | 5        | 8        |
+----------+----------+----------+
| N/A      | N/A      | N/A      |
+----------+----------+----------+
```

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_sheet1:
+----------+----------+----------+----------+
| Column 1 | Column 2 | Column 3 | Column 4 |
+----------+----------+----------+----------+
| 1        | 4        | 7        | 10       |
+----------+----------+----------+----------+
| 2        | 5        | 8        | 11       |
+----------+----------+----------+----------+
| 3        | 6        | 9        | 12       |
+----------+----------+----------+----------+
```

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_sheet1:
+----------+----------+----------+
| 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_sheet1:
```

(continues on next page)

(continued from previous page)

```
+----------+----------+----------+
| 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:

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_sheet1:
+-------+-----+-----+-----+
| Row 1 | 1   | 2   | 3   |
+-------+-----+-----+-----+
| Row 2 | 4   | 5   | 6   |
+-------+-----+-----+-----+
| Row 3 | 100 | 200 | 300 |
+-------+-----+-----+-----+
```

2.6.2 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.csv
'1,2,3\r\n3,4,5\r\n'
```

file type as its attributes

Since version 0.3.0, each supported file types became an attribute of the Sheet and Book class. What it means is that:

1. Read the content in memory
2. Set the content in memory

For example, after you have your Sheet and Book instance, you could access its content in a support file type by using its dot notation. The code in previous section could be rewritten as:

```
>>> import pyexcel
>>> content = "1,2,3\n3,4,5"
>>> sheet = pyexcel.Sheet()
>>> sheet.csv = content
>>> sheet.array
[[1, 2, 3], [3, 4, 5]]
```

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          content = request.files['excel'].read()
9          if sys.version_info[0] > 2:
10             # in order to support python 3
11             # have to decode bytes to str
12             content = content.decode('utf-8')
13         sheet = pe.get_sheet(file_type=extension, file_content=content)
14         # then use it as usual
15         sheet.name_columns_by_row(0)
16         # respond with a json
17         return jsonify({"result": sheet.dict})
18     return render_template('upload.html')
```

request.files['excel'] in line 4 holds the file object. line 5 finds out the file extension. line 13 obtains a sheet instance. line 15 uses the first row as data header. line 17 sends the json representation of the excel file back to client browser.

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      output = make_response(sheet.csv)
10     output.headers["Content-Disposition"] = "attachment; filename=export.csv"
11     output.headers["Content-type"] = "text/csv"
12     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

2.6.3 Sheet: Data conversion

How to obtain records from an excel sheet

Suppose you want to process the following excel data :

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
```

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 |
+---+---+---+
```

(continues on next page)

(continued from previous page)

```
| 7 | 8 | 9 |
+---+---+---+
```

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
```

How to get a dictionary from an excel sheet

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

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({str(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.

How to obtain a dictionary from a multiple sheet book

Suppose you have a multiple sheet book as the following:

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")
```

(continues on next page)

(continued from previous page)

```
>>> 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]]}
```

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():
```

(continues on next page)

(continued from previous page)

```
...     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.

How to import 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:

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=BirthRegister)
```

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 |
+------------+----+-------+--------+
```

(continues on next page)

(continued from previous page)

```
| 2015-02-03 | 1  | Adam  | 3.4    |
+------------+----+-------+--------+
| 2014-11-12 | 2  | Smith | 4.2    |
+------------+----+-------+--------+
```

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.

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:
+-------+--------+----------+
```

(continues on next page)

(continued from previous page)

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

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

Well, you write similar codes as before but you will need to use *save_book_as()* function.

2.6.4 Dot notation for data source

Since version 0.3.0, the data source becomes an attribute of the pyexcel native classes. All support data format is a dot notation away.

For sheet

Get content

```
>>> import pyexcel
>>> content = "1,2,3\n3,4,5"
>>> sheet = pyexcel.get_sheet(file_type="csv", file_content=content)
>>> sheet.tsv
'1\t2\t3\r\n3\t4\t5\r\n'
>>> print(sheet.simple)
csv:
-  -  -
1  2  3
3  4  5
-  -  -
```

What's more, you could as well set value to an attribute, for example:

```
>>> import pyexcel
>>> content = "1,2,3\n3,4,5"
>>> sheet = pyexcel.Sheet()
>>> sheet.csv = content
>>> sheet.array
[[1, 2, 3], [3, 4, 5]]
```

You can get the direct access to underneath stream object. In some situation, it is desired.

```
>>> stream = sheet.stream.tsv
```

The returned stream object has tsv formatted content for reading.

Set content

What you could further do is to set a memory stream of any supported file format to a sheet. For example:

```
>>> another_sheet = pyexcel.Sheet()
>>> another_sheet.xls = sheet.xls
```

(continues on next page)

(continued from previous page)

```
>>> another_sheet.content
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 3 | 4 | 5 |
+---+---+---+
```

Yet, it is possible assign a absolute url to an online excel file to an instance of *pyexcel.Sheet*.

```
>>> another_sheet.url = "https://github.com/pyexcel/pyexcel/raw/master/examples/basics/
↪multiple-sheets-example.xls"
>>> another_sheet.content
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 7 | 8 | 9 |
+---+---+---+
```

For book

The same dot notation is available to *pyexcel.Book* as well.

Get content

```
>>> book_dict = {
...     '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]
...         ],
...     'Sheet 1':
...         [
...             [1.0, 2.0, 3.0],
...             [4.0, 5.0, 6.0],
...             [7.0, 8.0, 9.0]
...         ]
... }
>>> book = pyexcel.get_book(bookdict=book_dict)
>>> book
Sheet 1:
+-----+-----+-----+
| 1.0 | 2.0 | 3.0 |
```

(continues on next page)

(continued from previous page)

```
+-----+-----+-----+
| 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 |
+-----+-----+-----+
>>> print(book.rst)
Sheet 1:
=  =  =
1  2  3
4  5  6
7  8  9
=  =  =
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
===  ===  ===
```

You can get the direct access to underneath stream object. In some situation, it is desired.

```
>>> stream = sheet.stream.plain
```

The returned stream object has the content formatted in plain format for further reading.

Set content

Surely, you could set content to an instance of *pyexcel.Book*.

```
>>> other_book = pyexcel.Book()
>>> other_book.bookdict = book_dict
>>> print(other_book.plain)
Sheet 1:
1  2  3
4  5  6
7  8  9
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
```

You can set via 'xls' attribute too.

```
>>> another_book = pyexcel.Book()
>>> another_book.xls = other_book.xls
>>> print(another_book.mediawiki)
Sheet 1:
{| class="wikitable" style="text-align: left;"
|+ <!-- caption -->
|-
| align="right"| 1 || align="right"| 2 || align="right"| 3
|-
| align="right"| 4 || align="right"| 5 || align="right"| 6
|-
| align="right"| 7 || align="right"| 8 || align="right"| 9
|}
Sheet 2:
{| class="wikitable" style="text-align: left;"
|+ <!-- caption -->
|-
| X || Y || Z
|-
| 1 || 2 || 3
|-
| 4 || 5 || 6
|}
Sheet 3:
{| class="wikitable" style="text-align: left;"
|+ <!-- caption -->
|-
| O || P || Q
|-
| 3 || 2 || 1
|-
| 4 || 3 || 2
|}
```

How about setting content via a url?

```
>>> another_book.url = "https://github.com/pyexcel/pyexcel/raw/master/examples/basics/
↪multiple-sheets-example.xls"
>>> another_book
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 |
+---+---+---+
```

Getters and Setters

You can pass on source specific parameters to getter and setter functions.

```
>>> content = "1-2-3\n3-4-5"
>>> sheet = pyexcel.Sheet()
>>> sheet.set_csv(content, delimiter="-")
>>> sheet.csv
'1,2,3\r\n3,4,5\r\n'
>>> sheet.get_csv(delimiter="|")
'1|2|3\r\n3|4|5\r\n'
```

2.6.5 Read partial data

When you are dealing with huge amount of data, e.g. 64GB, obviously you would not like to fill up your memory with those data. What you may want to do is, record data from Nth line, take M records and stop. And you only want to use your memory for the M records, not for beginning part nor for the tail part.

Hence partial read feature is developed to read partial data into memory for processing.

You can paginate by row, by column and by both, hence you dictate what portion of the data to read back. But remember only row limit features help you save memory. Let's you use this feature to record data from Nth column, take M number of columns and skip the rest. You are not going to reduce your memory footprint.

Why did not I see above benefit?

This feature depends heavily on the implementation details.

`pyexcel-xls`_ (xlrd), **`pyexcel-xlsx`**_ (openpyxl), **`pyexcel-ods`**_ (odfpy) and **`pyexcel-ods3`**_ (pyexcel-ezodf) will read all data into memory. Because xls, xlsx and ods file are effectively a zipped folder, all four will unzip the folder and read the content in xml format in **full**, so as to make sense of all details.

Hence, while the partial data is being returned, the memory consumption won't differ from reading the whole data back. Only after the partial data is returned, the memory consumption curve shall jump the cliff. So pagination code here only limits the data returned to your program.

With that said, **`pyexcel-xlsxr`**_, **`pyexcel-odsr`**_ and **`pyexcel-htmlr`**_ DOES read partial data into memory. Those three are implemented in such a way that they consume the xml(html) when needed. When they have read designated portion of the data, they stop, even if they are halfway through.

In addition, pyexcel's csv readers can read partial data into memory too.

Let's assume the following file is a huge csv file:

```
>>> import datetime
>>> import pyexcel as pe
>>> data = [
...     [1, 21, 31],
...     [2, 22, 32],
...     [3, 23, 33],
...     [4, 24, 34],
...     [5, 25, 35],
...     [6, 26, 36]
... ]
>>> pe.save_as(array=data, dest_file_name="your_file.csv")
```

And let's pretend to read partial data:

```
>>> pe.get_sheet(file_name="your_file.csv", start_row=2, row_limit=3)
your_file.csv:
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
```

And you could as well do the same for columns:

```
>>> pe.get_sheet(file_name="your_file.csv", start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 21 | 31 |
+----+----+
| 22 | 32 |
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
```

(continues on next page)

(continued from previous page)

```
| 25 | 35 |
+----+----+
| 26 | 36 |
+----+----+
```

Obviously, you could do both at the same time:

```
>>> pe.get_sheet(file_name="your_file.csv",
...     start_row=2, row_limit=3,
...     start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
```

The pagination support is available across all pyexcel plugins.

> **Note**
>
> No column pagination support for query sets as data source.

Formatting while transcoding a big data file

If you are transcoding a big data set, conventional formatting method would not help unless a on-demand free RAM is available. However, there is a way to minimize the memory footprint of pyexcel while the formatting is performed.

Let's continue from previous example. Suppose we want to transcode "your_file.csv" to "your_file.xls" but increase each element by 1.

What we can do is to define a row renderer function as the following:

```
>>> def increment_by_one(row):
...     for element in row:
...         yield element + 1
```

Then pass it onto save_as function using row_renderer:

```
>>> pe.isave_as(file_name="your_file.csv",
...             row_renderer=increment_by_one,
...             dest_file_name="your_file.xlsx")
```

> **Note**
>
> If the data content is from a generator, isave_as has to be used.

We can verify if it was done correctly:

```
>>> pe.get_sheet(file_name="your_file.xlsx")
your_file.csv:
+---+----+----+
| 2 | 22 | 32 |
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
| 6 | 26 | 36 |
+---+----+----+
| 7 | 27 | 37 |
+---+----+----+
```

2.6.6 Sheet: Data Access

Iterate a csv file

Here is the way to read the csv file and iterate through each row:

```
>>> sheet = pyexcel.get_sheet(file_name='tutorial.csv')
>>> for row in sheet:
...     print("%s: %s" % (row[0], row[1]))
Name: Age
Chu Chu: 10
Mo mo: 11
```

Often people wanted to use csv.Dict reader to read it because it has a header. Here is how you do it with pyexcel:

```
1 >>> sheet = pyexcel.get_sheet(file_name='tutorial.csv')
2 >>> sheet.name_columns_by_row(0)
3 >>> for row in sheet:
4 ...     print("%s: %s" % (row[0], row[1]))
5 Chu Chu: 10
6 Mo mo: 11
```

Line 2 remove the header from the actual content. The removed header can be used to access its columns using the name itself, for example:

```
>>> sheet.column['Age']
[10, 11]
```

Random access to individual cell

Top left corner of a sheet is (0, 0), meaning both row index and column index start from 0. To randomly access a cell of *Sheet* instance, two syntax are available:

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

This syntax helps you iterate the data by row and by column. If you use excel positions, the syntax below help you get the cell instantly without converting alphabet column index to integer:

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

Please note that with excel positions, top left corner is 'A1'.

For example: suppose you have the following data sheet,

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

```
>>> 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[3, 2])
8
>>> print(sheet["D3"])
6
>>> sheet[2, 3] = 10
>>> print(sheet[2, 3])
10
```

Note

In order to set a value to a cell, please use sheet[row_index, column_index] = new_value

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]
```

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
>>> sheet[1, "Y"] = 100
>>> print(sheet[1, "Y"])
100
```

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, 100, 8]
```

Hence access the same cell, this statement also works:

```
>>> sheet.column['Y'][1]
100
```

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]
100
```

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"])
100
>>> sheet["b", "Y"] = 200
>>> print(sheet["b", "Y"])
200
```

Note

When you have named your rows and columns, in order to set a value to a cell, please use sheet[row_name, column_name] = new_value

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:

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.

Reading a single sheet excel file

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

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:

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:

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
}
{
```

(continues on next page)

(continued from previous page)

```
'X':7
'Y':8
'Z':9
}
```

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:

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")
```

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'],
```

(continues on next page)

(continued from previous page)

```
...             [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

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
```

Work with data series in a single sheet

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

```
>>> 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:

```
>>> list(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:

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

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

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

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

```
>>> list(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()`:

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

2.6.7 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 exposed by `Column` and `Row`

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        |
```

(continues on next page)

(continued from previous page)

```
+----------+----------+----------+
| 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        |
+----------+----------+
```

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 |
+---+---+---+---+
```

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        |
+----------+----------+----------+
```

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)
>>> sheet
pyexcel sheet:
+-------+---+---+---+
| Row 1 | a | b | c |
+-------+---+---+---+
| Row 2 | e | f | g |
+-------+---+---+---+
| Row 3 | 1 | 2 | 3 |
+-------+---+---+---+
```

Then you can access rows by its name:

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

2.6.8 Sheet: Data 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)
>>> sheet.content
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+==========+==========+==========+
| 1        | 2        | 3        |
+----------+----------+----------+
| 4        | 5        | 6        |
+----------+----------+----------+
| 7        | 8        | 9        |
+----------+----------+----------+
```

Filter out some data

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

```
>>> sheet.filter(row_indices=[0, 2])
>>> sheet.content
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+==========+==========+==========+
| 4        | 5        | 6        |
+----------+----------+----------+
```

Let's try to further filter out even columns:

```
>>> sheet.filter(column_indices=[1])
>>> sheet.content
+----------+----------+
| Column 1 | Column 3 |
+==========+==========+
| 4        | 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

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]])
```

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_index, row):
...     result = [element for element in row if element != '']
...     return len(result)==0
```

And then apply the filter on the sheet:

```
>>> del sheet.row[filter_row]
>>> sheet
pyexcel sheet:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
```

2.6.9 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. use *`format()`* to apply formatter immediately.

> **Note**
>
> **int**, **float** and **datetime** values are automatically detected in **csv** files since **pyexcel** version 0.2.2

Convert a column of numbers to strings

Suppose you have the following data:

```
>>> import pyexcel
>>> data = [
...     ["userid","name"],
...     [10120,"Adam"],
...     [10121,"Bella"],
...     [10122,"Cedar"]
... ]
>>> sheet = pyexcel.Sheet(data)
>>> sheet.name_columns_by_row(0)
>>> sheet.column["userid"]
[10120, 10121, 10122]
```

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

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

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 " ":

```
>>> data = [
...     ["        Version", "        Comments", "       Author  "],
...     ["  v0.0.1       ", " Release versions","            Eda"],
...     ["  v0.0.2  ", "Useful updates    ", "   Freud"]
... ]
>>> sheet = pyexcel.Sheet(data)
>>> sheet.content
+-----------------+------------------------------+----------------------+
|         Version |         Comments             |        Author   |
+-----------------+------------------------------+----------------------+
|   v0.0.1        |  Release versions            |             Eda |
+-----------------+------------------------------+----------------------+
|   v0.0.2   | Useful updates     |    Freud        |
+-----------------+------------------------------+----------------------+
```

Now try to create a custom cleanse function:

```
.. code-block:: python
```

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

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

```
.. code-block:: python
```

```
>>> sheet.map(cleanse_func)
```

So in the end, you get this:

```
>>> sheet.content
+---------+------------------+--------+
| Version | Comments         | Author |
+---------+------------------+--------+
| v0.0.1  | Release versions | Eda    |
+---------+------------------+--------+
| v0.0.2  | Useful updates   | Freud  |
+---------+------------------+--------+
```

2.6.10 Book: Sheet operations

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 |
+---+---+---+
```

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 individual 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"]
```

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

How do I read a book, process 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

Alternatively, you could use *save_book_as()* function

```
>>> pyexcel.save_book_as(bookdict=content, dest_file_name="myfile.csv")
```

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

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:
+-----+-----+-----+
| 1.0 | 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 |
+-----+-----+-----+
```

2.7 Cook book

2.7.1 Recipes

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

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

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")
```

(continues on next page)

(continued from previous page)

```
>>> pyexcel.get_sheet(file_name="output.xls")
pyexcel sheet:
+-------+----+----+----+
| Row 1 | 11 | 12 | 13 |
+-------+----+----+----+
| Row 2 | 4  | 5  | 6  |
+-------+----+----+----+
| Row 3 | 7  | 8  | 9  |
+-------+----+----+----+
```

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		

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
>>> 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)
>>> del sheet1.column[1, 3, 5]
>>> del sheet2.column[0, 1]
>>> merge_two_readers(sheet1, sheet2, "output.xls")
```

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

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")
```

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

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

2.7.2 Loading from other sources

Get back into pyexcel

list

```
>>> import pyexcel as p
>>> two_dimensional_list = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
>>> sheet = p.get_sheet(array=two_dimensional_list)
```

(continues on next page)

(continued from previous page)

```
>>> sheet
pyexcel_sheet1:
+---+----+----+----+
| 1 | 2  | 3  | 4  |
+---+----+----+----+
| 5 | 6  | 7  | 8  |
+---+----+----+----+
| 9 | 10 | 11 | 12 |
+---+----+----+----+
```

dict

```
>>> a_dictionary_of_key_value_pair = {
...     "IE": 0.2,
...     "Firefox": 0.3
... }
>>> sheet = p.get_sheet(adict=a_dictionary_of_key_value_pair)
>>> sheet
pyexcel_sheet1:
+---------+-----+
| Firefox | IE  |
+---------+-----+
| 0.3     | 0.2 |
+---------+-----+
```

```
>>> a_dictionary_of_one_dimensional_arrays = {
...     "Column 1": [1, 2, 3, 4],
...     "Column 2": [5, 6, 7, 8],
...     "Column 3": [9, 10, 11, 12],
... }
>>> sheet = p.get_sheet(adict=a_dictionary_of_one_dimensional_arrays)
>>> sheet
pyexcel_sheet1:
+----------+----------+----------+
| Column 1 | Column 2 | Column 3 |
+----------+----------+----------+
| 1        | 5        | 9        |
+----------+----------+----------+
| 2        | 6        | 10       |
+----------+----------+----------+
| 3        | 7        | 11       |
+----------+----------+----------+
| 4        | 8        | 12       |
+----------+----------+----------+
```

records

```
>>> a_list_of_dictionaries = [
...     {
...         "Name": 'Adam',
```

(continues on next page)

(continued from previous page)

```
...         "Age": 28
...     },
...     {
...         "Name": 'Beatrice',
...         "Age": 29
...     },
...     {
...         "Name": 'Ceri',
...         "Age": 30
...     },
...     {
...         "Name": 'Dean',
...         "Age": 26
...     }
... ]
>>> sheet = p.get_sheet(records=a_list_of_dictionaries)
>>> sheet
pyexcel_sheet1:
+-----+----------+
| Age | Name     |
+-----+----------+
| 28  | Adam     |
+-----+----------+
| 29  | Beatrice |
+-----+----------+
| 30  | Ceri     |
+-----+----------+
| 26  | Dean     |
+-----+----------+
```

book dict

```
>>> 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]
...         ]
```

(continues on next page)

(continued from previous page)

```
...  }
>>> book = p.get_book(bookdict=a_dictionary_of_two_dimensional_arrays)
>>> book
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 |
+-----+-----+-----+
```

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 |
+---+---+---+
```

For sheet

Get content

```
>>> another_sheet = p.Sheet()
>>> another_sheet.url = "https://github.com/pyexcel/pyexcel/raw/master/examples/basics/
↪multiple-sheets-example.xls"
>>> another_sheet.content
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 4 | 5 | 6 |
```

(continues on next page)

(continued from previous page)

```
+---+---+---+
| 7 | 8 | 9 |
+---+---+---+
```

For book

How about setting content via a url?

```
>>> another_book = p.Book()
>>> another_book.url = "https://github.com/pyexcel/pyexcel/raw/master/examples/basics/
↪multiple-sheets-example.xls"
>>> another_book
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 |
+---+---+---+
```

2.8 Real world cases

2.8.1 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. Python convert csv to xlsx
6. How to read data from excel and set data type
7. Remove or keep specific columns in csv file

8. How can I put a CSV file in an array?

2.8.2 How to inject csv data to database

Here is real case in the stack-overflow. Due to the author's ignorance, the user would like to have the code in matlab than Python. Hence, I am sharing my pyexcel solution here.

Problem definition

Here is my CSV file:

```
PDB_Id      123442  234335  234336  3549867
a001        6       0       0       8
b001        4       2       0       0
c003        0       0       0       5
```

I want to put this data in a MYSQL table in the form:

```
PROTEIN_ID  PROTEIN_KEY     VALUE_OF_KEY
a001            123442          6
a001            234335          0
a001            234336          0
a001            3549867         8
b001            123442          4
b001            234335          2
b001            234336          0
b001            234336          0
c003            123442          0
c003            234335          0
c003            234336          0
c003            3549867         5
```

I have created table with the following code:

```
sql = """CREATE TABLE ALLPROTEINS (
        Protein_ID CHAR(20),
        PROTEIN_KEY INT ,
        VALUE_OF_KEY INT
        )"""
```

I need the code for insert.

Pyexcel solution

If you could insert an id field to act as the primary key, it can be mapped using sqlalchemy's ORM:

```
$ sqlite3 /tmp/stack2.db
sqlite> CREATE TABLE ALLPROTEINS (
   ...>          ID INT,
   ...>          Protein_ID CHAR(20),
   ...>          PROTEIN_KEY INT,
   ...>          VALUE_OF_KEY INT
   ...>          );
```

Here is the data mapping script vis sqlalchemy:

```
>>> # mapping your database via sqlalchemy
>>> from sqlalchemy import create_engine
>>> from sqlalchemy.ext.declarative import declarative_base
>>> from sqlalchemy import Column, Integer, String
>>> from sqlalchemy.orm import sessionmaker
>>> # checkout http://docs.sqlalchemy.org/en/latest/dialects/index.html
>>> # for a different database server
>>> engine = create_engine("sqlite:////tmp/stack2.db")
>>> Base = declarative_base()
>>> class Proteins(Base):
...     __tablename__ = 'ALLPROTEINS'
...     id = Column(Integer, primary_key=True, autoincrement=True) # <-- appended field
...     protein_id = Column(String(20))
...     protein_key = Column(Integer)
...     value_of_key = Column(Integer)
>>> Session = sessionmaker(bind=engine)
>>>
```

Here is the short script to get data inserted into the database:

```
>>> import pyexcel as p
>>> from itertools import product
>>> # data insertion code starts here
>>> sheet = p.get_sheet(file_name="csv-to-mysql-in-matlab-code.csv", delimiter='\t')
>>> sheet.name_columns_by_row(0)
>>> sheet.name_rows_by_column(0)
>>> print(sheet)
csv-to-mysql-in-matlab-code.csv:
+------+--------+--------+--------+---------+
|      | 123442 | 234335 | 234336 | 3549867 |
+======+========+========+========+=========+
| a001 | 6      | 0      | 0      | 8       |
+------+--------+--------+--------+---------+
| b001 | 4      | 2      | 0      | 0       |
+------+--------+--------+--------+---------+
| c003 | 0      | 0      | 0      | 5       |
+------+--------+--------+--------+---------+
>>> results = []
>>> for protein_id, protein_key in product(sheet.rownames, sheet.colnames):
...     results.append([protein_id, protein_key, sheet[str(protein_id), protein_key]])
>>>
>>> sheet2 = p.get_sheet(array=results)
>>> sheet2.colnames = ['protein_id', 'protein_key', 'value_of_key']
>>> print(sheet2)
pyexcel_sheet1:
+------------+-------------+--------------+
| protein_id | protein_key | value_of_key |
+============+=============+==============+
| a001       | 123442      | 6            |
+------------+-------------+--------------+
| a001       | 234335      | 0            |
+------------+-------------+--------------+
| a001       | 234336      | 0            |
```

(continues on next page)

(continued from previous page)

```
+------------+-------------+--------------+
| a001       | 3549867     | 8            |
+------------+-------------+--------------+
| b001       | 123442      | 4            |
+------------+-------------+--------------+
| b001       | 234335      | 2            |
+------------+-------------+--------------+
| b001       | 234336      | 0            |
+------------+-------------+--------------+
| b001       | 3549867     | 0            |
+------------+-------------+--------------+
| c003       | 123442      | 0            |
+------------+-------------+--------------+
| c003       | 234335      | 0            |
+------------+-------------+--------------+
| c003       | 234336      | 0            |
+------------+-------------+--------------+
| c003       | 3549867     | 5            |
+------------+-------------+--------------+
>>> sheet2.save_to_database(session=Session(), table=Proteins)
```

Here is the data inserted:

```
$ sqlite3 /tmp/stack2.db
sqlite> select * from allproteins
   ...> ;
|a001|123442|6
|a001|234335|0
|a001|234336|0
|a001|3549867|8
|b001|123442|4
|b001|234335|2
|b001|234336|0
|b001|234336|0
|c003|123442|0
|c003|234335|0
|c003|234336|0
|c003|3549867|5
```

2.9 API documentation

2.9.1 API Reference

This is intended for users of pyexcel.

Signature functions

Obtaining data from excel file

get_array(**keywords)	Obtain an array from an excel source

continues on next page

Table 9 – continued from previous page

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
iget_book(**keywords)	Get an instance of BookStream from an excel source
iget_array(**keywords)	Obtain a generator of a two-dimensional array from an excel source
iget_records([custom_headers])	Obtain a generator of a list of records from an excel source
free_resources()	Close file handles opened by signature functions that starts with 'i'

pyexcel.get_array

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

Obtain an array from an excel source

It accepts the same parameters as *get_sheet()* but return an array instead.

Not all parameters are needed. Here is a table

source	parameters
loading from file	file_name, sheet_name, keywords
loading from string	file_content, file_type, sheet_name, keywords
loading from stream	file_stream, file_type, 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
loading from an url	url

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 *file_content* or *file_stream*

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

sheet_name :
sheet name. if sheet_name is not given, the default sheet at index 0 is loaded

start_row
[int] defaults to 0. It allows you to skip rows at the beginning

row_limit: int
defaults to -1, meaning till the end of the whole sheet. It allows you to skip the tailing rows.

start_column
[int] defaults to 0. It allows you to skip columns on your left hand side

column_limit: int
defaults to -1, meaning till the end of the columns. It allows you to skip the tailing columns.

skip_row_func:
It allows you to write your own row skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_column_func:
It allows you to write your own column skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_empty_rows: bool
Defaults to False. Toggle it to True if the rest of empty rows are useless, but it does affect the number of rows.

row_renderer:
You could choose to write a custom row renderer when the data is being read.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

Parameters related to xls file format:
Please note the following parameters apply to pyexcel-xls. more details can be found in `xlrd.open_workbook()`

logfile:
An open file to which messages and diagnostics are written.

verbosity:
Increases the volume of trace material written to the logfile.

use_mmap:
Whether to use the mmap module is determined heuristically. Use this arg to override the result.

Current heuristic: mmap is used if it exists.

encoding_override:
Used to overcome missing or bad codepage information in older-version files.

formatting_info:
The default is False, which saves memory.

When True, formatting information will be read from the spreadsheet file. This provides all cells, including empty and blank cells. Formatting information is available for each cell.

ragged_rows:
The default of False means all rows are padded out with empty cells so that all rows have the same size as found in ncols.

True means that there are no empty cells at the ends of rows. This can result in substantial memory savings if rows are of widely varying sizes. See also the row_len() method.

pyexcel.get_dict

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

Obtain a dictionary from an excel source

It accepts the same parameters as *get_sheet()* but return a dictionary instead.

Specifically: name_columns_by_row : specify a row to be a dictionary key. It is default to 0 or first row.

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

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

Examples on start_row, start_column

Let's assume the following file is a huge csv file:

```
>>> import datetime
>>> import pyexcel as pe
>>> data = [
...     [1, 21, 31],
...     [2, 22, 32],
...     [3, 23, 33],
...     [4, 24, 34],
...     [5, 25, 35],
...     [6, 26, 36]
... ]
>>> pe.save_as(array=data, dest_file_name="your_file.csv")
```

And let's pretend to read partial data:

```
>>> pe.get_sheet(file_name="your_file.csv", start_row=2, row_limit=3)
your_file.csv:
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
```

And you could as well do the same for columns:

```
>>> pe.get_sheet(file_name="your_file.csv", start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 21 | 31 |
+----+----+
| 22 | 32 |
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
| 26 | 36 |
+----+----+
```

Obvious, you could do both at the same time:

```
>>> pe.get_sheet(file_name="your_file.csv",
...     start_row=2, row_limit=3,
...     start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
```

The pagination support is available across all pyexcel plugins.

Note

No column pagination support for query sets as data source.

Formatting while transcoding a big data file

If you are transcoding a big data set, conventional formatting method would not help unless a on-demand free RAM is available. However, there is a way to minimize the memory footprint of pyexcel while the formatting is performed.

Let's continue from previous example. Suppose we want to transcode "your_file.csv" to "your_file.xls" but increase each element by 1.

What we can do is to define a row renderer function as the following:

```
>>> def increment_by_one(row):
...     for element in row:
...         yield element + 1
```

Then pass it onto save_as function using row_renderer:

```
>>> pe.isave_as(file_name="your_file.csv",
...             row_renderer=increment_by_one,
...             dest_file_name="your_file.xlsx")
```

Note

If the data content is from a generator, isave_as has to be used.

We can verify if it was done correctly:

```
>>> pe.get_sheet(file_name="your_file.xlsx")
your_file.csv:
+---+----+----+
| 2 | 22 | 32 |
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
| 6 | 26 | 36 |
+---+----+----+
| 7 | 27 | 37 |
+---+----+----+
```

Not all parameters are needed. Here is a table

source	parameters
loading from file	file_name, sheet_name, keywords
loading from string	file_content, file_type, sheet_name, keywords
loading from stream	file_stream, file_type, 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
loading from an url	url

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 *file_content* or *file_stream*

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

sheet_name :
sheet name. if sheet_name is not given, the default sheet at index 0 is loaded

start_row
[int] defaults to 0. It allows you to skip rows at the beginning

row_limit: int
defaults to -1, meaning till the end of the whole sheet. It allows you to skip the tailing rows.

start_column
[int] defaults to 0. It allows you to skip columns on your left hand side

column_limit: int
defaults to -1, meaning till the end of the columns. It allows you to skip the tailing columns.

skip_row_func:
It allows you to write your own row skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_column_func:
It allows you to write your own column skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_empty_rows: bool
Defaults to False. Toggle it to True if the rest of empty rows are useless, but it does affect the number of rows.

row_renderer:
You could choose to write a custom row renderer when the data is being read.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

Parameters related to xls file format:
Please note the following parameters apply to pyexcel-xls. more details can be found in `xlrd.open_workbook()`

logfile:
An open file to which messages and diagnostics are written.

verbosity:
Increases the volume of trace material written to the logfile.

use_mmap:
Whether to use the mmap module is determined heuristically. Use this arg to override the result.

Current heuristic: mmap is used if it exists.

encoding_override:
Used to overcome missing or bad codepage information in older-version files.

formatting_info:
The default is False, which saves memory.

When True, formatting information will be read from the spreadsheet file. This provides all cells, including empty and blank cells. Formatting information is available for each cell.

ragged_rows:
The default of False means all rows are padded out with empty cells so that all rows have the same size as found in ncols.

True means that there are no empty cells at the ends of rows. This can result in substantial memory savings if rows are of widely varying sizes. See also the row_len() method.

pyexcel.get_records

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

Obtain a list of records from an excel source

It accepts the same parameters as *get_sheet()* but return a list of dictionary(records) instead.

Specifically: name_columns_by_row : specify a row to be a dictionary key. It is default to 0 or first row.

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

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

Examples on start_row, start_column

Let's assume the following file is a huge csv file:

```
>>> import datetime
>>> import pyexcel as pe
>>> data = [
...     [1, 21, 31],
...     [2, 22, 32],
...     [3, 23, 33],
...     [4, 24, 34],
...     [5, 25, 35],
...     [6, 26, 36]
... ]
>>> pe.save_as(array=data, dest_file_name="your_file.csv")
```

And let's pretend to read partial data:

```
>>> pe.get_sheet(file_name="your_file.csv", start_row=2, row_limit=3)
your_file.csv:
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
```

(continues on next page)

(continued from previous page)

```
| 5 | 25 | 35 |
+---+----+----+
```

And you could as well do the same for columns:

```
>>> pe.get_sheet(file_name="your_file.csv", start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 21 | 31 |
+----+----+
| 22 | 32 |
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
| 26 | 36 |
+----+----+
```

Obvious, you could do both at the same time:

```
>>> pe.get_sheet(file_name="your_file.csv",
...     start_row=2, row_limit=3,
...     start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
```

The pagination support is available across all pyexcel plugins.

Note

No column pagination support for query sets as data source.

Formatting while transcoding a big data file

If you are transcoding a big data set, conventional formatting method would not help unless a on-demand free RAM is available. However, there is a way to minimize the memory footprint of pyexcel while the formatting is performed.

Let's continue from previous example. Suppose we want to transcode "your_file.csv" to "your_file.xls" but increase each element by 1.

What we can do is to define a row renderer function as the following:

```
>>> def increment_by_one(row):
...     for element in row:
...         yield element + 1
```

Then pass it onto save_as function using row_renderer:

```
>>> pe.isave_as(file_name="your_file.csv",
...             row_renderer=increment_by_one,
...             dest_file_name="your_file.xlsx")
```

Note

If the data content is from a generator, isave_as has to be used.

We can verify if it was done correctly:

```
>>> pe.get_sheet(file_name="your_file.xlsx")
your_file.csv:
+---+----+----+
| 2 | 22 | 32 |
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
| 6 | 26 | 36 |
+---+----+----+
| 7 | 27 | 37 |
+---+----+----+
```

Not all parameters are needed. Here is a table

source	parameters
loading from file	file_name, sheet_name, keywords
loading from string	file_content, file_type, sheet_name, keywords
loading from stream	file_stream, file_type, 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
loading from an url	url

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 *file_content* or *file_stream*

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

sheet_name :
sheet name. if sheet_name is not given, the default sheet at index 0 is loaded

start_row
[int] defaults to 0. It allows you to skip rows at the beginning

row_limit: int
defaults to -1, meaning till the end of the whole sheet. It allows you to skip the tailing rows.

start_column
[int] defaults to 0. It allows you to skip columns on your left hand side

column_limit: int
defaults to -1, meaning till the end of the columns. It allows you to skip the tailing columns.

skip_row_func:
It allows you to write your own row skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_column_func:
It allows you to write your own column skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_empty_rows: bool
Defaults to False. Toggle it to True if the rest of empty rows are useless, but it does affect the number of rows.

row_renderer:
You could choose to write a custom row renderer when the data is being read.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

Parameters related to xls file format:
Please note the following parameters apply to pyexcel-xls. more details can be found in `xlrd.open_workbook()`

logfile:
An open file to which messages and diagnostics are written.

verbosity:
Increases the volume of trace material written to the logfile.

use_mmap:
Whether to use the mmap module is determined heuristically. Use this arg to override the result.

Current heuristic: mmap is used if it exists.

encoding_override:
Used to overcome missing or bad codepage information in older-version files.

formatting_info:
The default is False, which saves memory.

When True, formatting information will be read from the spreadsheet file. This provides all cells, including empty and blank cells. Formatting information is available for each cell.

ragged_rows:
The default of False means all rows are padded out with empty cells so that all rows have the same size as found in ncols.

True means that there are no empty cells at the ends of rows. This can result in substantial memory savings if rows are of widely varying sizes. See also the row_len() method.

pyexcel.get_book_dict

`pyexcel.get_book_dict(**keywords)`

Obtain a dictionary of two dimensional arrays

It accepts the same parameters as *`get_book()`* but return a dictionary instead.

Here is a table of parameters:

source	parameters
loading from file	file_name, keywords
loading from string	file_content, file_type, keywords
loading from stream	file_stream, file_type, keywords
loading from sql	session, tables
loading from django models	models
loading from dictionary	bookdict
loading from an url	url

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

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 *file_content* or *file_stream*

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

sheets:
a list of mixed sheet names and sheet indices to be read. This is done to keep Pandas compactibility. With this parameter, more than one sheet can be read and you have the control to read the sheets of your interest instead of all available sheets.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

pyexcel.get_book

`pyexcel.get_book(**keywords)`

Get an instance of *Book* from an excel source

Here is a table of parameters:

source	parameters
loading from file	file_name, keywords
loading from string	file_content, file_type, keywords
loading from stream	file_stream, file_type, keywords
loading from sql	session, tables
loading from django models	models
loading from dictionary	bookdict
loading from an url	url

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

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 *file_content* or *file_stream*

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

sheets:
a list of mixed sheet names and sheet indices to be read. This is done to keep Pandas compactibility. With this parameter, more than one sheet can be read and you have the control to read the sheets of your interest instead of all available sheets.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

pyexcel.get_sheet

pyexcel.**get_sheet**(***keywords*)

Get an instance of *Sheet* from an excel source

Examples on start_row, start_column

Let's assume the following file is a huge csv file:

```
>>> import datetime
>>> import pyexcel as pe
>>> data = [
...     [1, 21, 31],
...     [2, 22, 32],
...     [3, 23, 33],
...     [4, 24, 34],
...     [5, 25, 35],
...     [6, 26, 36]
... ]
>>> pe.save_as(array=data, dest_file_name="your_file.csv")
```

And let's pretend to read partial data:

```
>>> pe.get_sheet(file_name="your_file.csv", start_row=2, row_limit=3)
your_file.csv:
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
```

And you could as well do the same for columns:

```
>>> pe.get_sheet(file_name="your_file.csv", start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 21 | 31 |
+----+----+
| 22 | 32 |
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
```

(continues on next page)

(continued from previous page)

```
| 26 | 36 |
+----+----+
```

Obvious, you could do both at the same time:

```
>>> pe.get_sheet(file_name="your_file.csv",
...     start_row=2, row_limit=3,
...     start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
```

The pagination support is available across all pyexcel plugins.

Note

No column pagination support for query sets as data source.

Formatting while transcoding a big data file

If you are transcoding a big data set, conventional formatting method would not help unless a on-demand free RAM is available. However, there is a way to minimize the memory footprint of pyexcel while the formatting is performed.

Let's continue from previous example. Suppose we want to transcode "your_file.csv" to "your_file.xls" but increase each element by 1.

What we can do is to define a row renderer function as the following:

```
>>> def increment_by_one(row):
...     for element in row:
...         yield element + 1
```

Then pass it onto save_as function using row_renderer:

```
>>> pe.isave_as(file_name="your_file.csv",
...             row_renderer=increment_by_one,
...             dest_file_name="your_file.xlsx")
```

Note

If the data content is from a generator, isave_as has to be used.

We can verify if it was done correctly:

```
>>> pe.get_sheet(file_name="your_file.xlsx")
your_file.csv:
```

(continues on next page)

(continued from previous page)

```
+---+----+----+
| 2 | 22 | 32 |
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
| 6 | 26 | 36 |
+---+----+----+
| 7 | 27 | 37 |
+---+----+----+
```

Not all parameters are needed. Here is a table

source	parameters
loading from file	file_name, sheet_name, keywords
loading from string	file_content, file_type, sheet_name, keywords
loading from stream	file_stream, file_type, 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
loading from an url	url

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 *file_content* or *file_stream*

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

sheet_name :
sheet name. if sheet_name is not given, the default sheet at index 0 is loaded

start_row
[int] defaults to 0. It allows you to skip rows at the beginning

row_limit: int
defaults to -1, meaning till the end of the whole sheet. It allows you to skip the tailing rows.

start_column
[int] defaults to 0. It allows you to skip columns on your left hand side

column_limit: int
defaults to -1, meaning till the end of the columns. It allows you to skip the tailing columns.

skip_row_func:
It allows you to write your own row skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_column_func:
It allows you to write your own column skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_empty_rows: bool
Defaults to False. Toggle it to True if the rest of empty rows are useless, but it does affect the number of rows.

row_renderer:
You could choose to write a custom row renderer when the data is being read.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

Parameters related to xls file format:
Please note the following parameters apply to pyexcel-xls. more details can be found in `xlrd.open_workbook()`

logfile:
An open file to which messages and diagnostics are written.

verbosity:
Increases the volume of trace material written to the logfile.

use_mmap:
Whether to use the mmap module is determined heuristically. Use this arg to override the result.

Current heuristic: mmap is used if it exists.

encoding_override:
Used to overcome missing or bad codepage information in older-version files.

formatting_info:
The default is False, which saves memory.

When True, formatting information will be read from the spreadsheet file. This provides all cells, including empty and blank cells. Formatting information is available for each cell.

ragged_rows:
The default of False means all rows are padded out with empty cells so that all rows have the same size as found in ncols.

True means that there are no empty cells at the ends of rows. This can result in substantial memory savings if rows are of widely varying sizes. See also the row_len() method.

pyexcel.iget_book

pyexcel.**iget_book**(***keywords*)

Get an instance of BookStream from an excel source

First use case is to get all sheet names without extracting the sheets into memory.

Here is a table of parameters:

source	parameters
loading from file	file_name, keywords
loading from string	file_content, file_type, keywords
loading from stream	file_stream, file_type, keywords
loading from sql	session, tables
loading from django models	models
loading from dictionary	bookdict
loading from an url	url

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

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 *file_content* or *file_stream*

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

sheets:
a list of mixed sheet names and sheet indices to be read. This is done to keep Pandas compactibility. With

this parameter, more than one sheet can be read and you have the control to read the sheets of your interest instead of all available sheets.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

When you use this function to work on physical files, this function will leave its file handle open. When you finish the operation on its data, you need to call *pyexcel.free_resources()* to close file handle(s).

for csv, csvz file formats, file handles will be left open. for xls, ods file formats, the file is read all into memory and is close afterwards. for xlsx, file handles will be left open in python 2.7 - 3.5 by pyexcel-xlsx(openpyxl). In other words, pyexcel-xls, pyexcel-ods, pyexcel-ods3 won't leak file handles.

pyexcel.iget_array

pyexcel.**iget_array**(***keywords*)

Obtain a generator of a two-dimensional array from an excel source

It is similar to *pyexcel.get_array()* but it has less memory footprint.

Not all parameters are needed. Here is a table

source	parameters
loading from file	file_name, sheet_name, keywords
loading from string	file_content, file_type, sheet_name, keywords
loading from stream	file_stream, file_type, 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
loading from an url	url

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 *file_content* or *file_stream*

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

sheet_name :
sheet name. if sheet_name is not given, the default sheet at index 0 is loaded

start_row
[int] defaults to 0. It allows you to skip rows at the beginning

row_limit: int
defaults to -1, meaning till the end of the whole sheet. It allows you to skip the tailing rows.

start_column
[int] defaults to 0. It allows you to skip columns on your left hand side

column_limit: int
defaults to -1, meaning till the end of the columns. It allows you to skip the tailing columns.

skip_row_func:
It allows you to write your own row skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_column_func:
It allows you to write your own column skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_empty_rows: bool
Defaults to False. Toggle it to True if the rest of empty rows are useless, but it does affect the number of rows.

row_renderer:
You could choose to write a custom row renderer when the data is being read.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

Parameters related to xls file format:
Please note the following parameters apply to pyexcel-xls. more details can be found in `xlrd.open_workbook()`

logfile:
An open file to which messages and diagnostics are written.

verbosity:
Increases the volume of trace material written to the logfile.

use_mmap:
Whether to use the mmap module is determined heuristically. Use this arg to override the result.

Current heuristic: mmap is used if it exists.

encoding_override:
Used to overcome missing or bad codepage information in older-version files.

formatting_info:
The default is False, which saves memory.

When True, formatting information will be read from the spreadsheet file. This provides all cells, including empty and blank cells. Formatting information is available for each cell.

ragged_rows:
The default of False means all rows are padded out with empty cells so that all rows have the same size as found in ncols.

True means that there are no empty cells at the ends of rows. This can result in substantial memory savings if rows are of widely varying sizes. See also the row_len() method.

When you use this function to work on physical files, this function will leave its file handle open. When you finish the operation on its data, you need to call *pyexcel.free_resources()* to close file handle(s).

for csv, csvz file formats, file handles will be left open. for xls, ods file formats, the file is read all into memory and is close afterwards. for xlsx, file handles will be left open in python 2.7 - 3.5 by pyexcel-xlsx(openpyxl). In other words, pyexcel-xls, pyexcel-ods, pyexcel-ods3 won't leak file handles.

pyexcel.iget_records

pyexcel.**iget_records**(*custom_headers=None*, ***keywords*)

Obtain a generator of a list of records from an excel source

It is similar to *pyexcel.get_records()* but it has less memory footprint but requires the headers to be in the first row. And the data matrix should be of equal length. It should consume less memory and should work well with large files.

Examples on start_row, start_column

Let's assume the following file is a huge csv file:

```
>>> import datetime
>>> import pyexcel as pe
>>> data = [
...     [1, 21, 31],
...     [2, 22, 32],
...     [3, 23, 33],
...     [4, 24, 34],
...     [5, 25, 35],
...     [6, 26, 36]
... ]
>>> pe.save_as(array=data, dest_file_name="your_file.csv")
```

And let's pretend to read partial data:

```
>>> pe.get_sheet(file_name="your_file.csv", start_row=2, row_limit=3)
your_file.csv:
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
```

And you could as well do the same for columns:

```
>>> pe.get_sheet(file_name="your_file.csv", start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 21 | 31 |
+----+----+
| 22 | 32 |
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
```

(continues on next page)

(continued from previous page)

```
+----+----+
| 25 | 35 |
+----+----+
| 26 | 36 |
+----+----+
```

Obvious, you could do both at the same time:

```
>>> pe.get_sheet(file_name="your_file.csv",
...     start_row=2, row_limit=3,
...     start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
```

The pagination support is available across all pyexcel plugins.

Note

No column pagination support for query sets as data source.

Formatting while transcoding a big data file

If you are transcoding a big data set, conventional formatting method would not help unless a on-demand free RAM is available. However, there is a way to minimize the memory footprint of pyexcel while the formatting is performed.

Let's continue from previous example. Suppose we want to transcode "your_file.csv" to "your_file.xls" but increase each element by 1.

What we can do is to define a row renderer function as the following:

```
>>> def increment_by_one(row):
...     for element in row:
...         yield element + 1
```

Then pass it onto save_as function using row_renderer:

```
>>> pe.isave_as(file_name="your_file.csv",
...             row_renderer=increment_by_one,
...             dest_file_name="your_file.xlsx")
```

Note

If the data content is from a generator, isave_as has to be used.

We can verify if it was done correctly:

```
>>> pe.get_sheet(file_name="your_file.xlsx")
your_file.csv:
+---+----+----+
| 2 | 22 | 32 |
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
| 6 | 26 | 36 |
+---+----+----+
| 7 | 27 | 37 |
+---+----+----+
```

Not all parameters are needed. Here is a table

source	parameters
loading from file	file_name, sheet_name, keywords
loading from string	file_content, file_type, sheet_name, keywords
loading from stream	file_stream, file_type, 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
loading from an url	url

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 *file_content* or *file_stream*

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

sheet_name :
sheet name. if sheet_name is not given, the default sheet at index 0 is loaded

start_row
[int] defaults to 0. It allows you to skip rows at the beginning

row_limit: int
defaults to -1, meaning till the end of the whole sheet. It allows you to skip the tailing rows.

start_column
[int] defaults to 0. It allows you to skip columns on your left hand side

column_limit: int
defaults to -1, meaning till the end of the columns. It allows you to skip the tailing columns.

skip_row_func:
It allows you to write your own row skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_column_func:
It allows you to write your own column skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_empty_rows: bool
Defaults to False. Toggle it to True if the rest of empty rows are useless, but it does affect the number of rows.

row_renderer:
You could choose to write a custom row renderer when the data is being read.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

Parameters related to xls file format:
Please note the following parameters apply to pyexcel-xls. more details can be found in `xlrd.open_workbook()`

logfile:
An open file to which messages and diagnostics are written.

verbosity:
Increases the volume of trace material written to the logfile.

use_mmap:
Whether to use the mmap module is determined heuristically. Use this arg to override the result.

Current heuristic: mmap is used if it exists.

encoding_override:
Used to overcome missing or bad codepage information in older-version files.

formatting_info:
The default is False, which saves memory.

When True, formatting information will be read from the spreadsheet file. This provides all cells, including empty and blank cells. Formatting information is available for each cell.

ragged_rows:
The default of False means all rows are padded out with empty cells so that all rows have the same size as found in ncols.

True means that there are no empty cells at the ends of rows. This can result in substantial memory savings if rows are of widely varying sizes. See also the row_len() method.

When you use this function to work on physical files, this function will leave its file handle open. When you finish the operation on its data, you need to call *pyexcel.free_resources()* to close file handle(s).

for csv, csvz file formats, file handles will be left open. for xls, ods file formats, the file is read all into memory and is close afterwards. for xlsx, file handles will be left open in python 2.7 - 3.5 by pyexcel-xlsx(openpyxl). In other words, pyexcel-xls, pyexcel-ods, pyexcel-ods3 won't leak file handles.

pyexcel.free_resources

pyexcel.**free_resources**()

Close file handles opened by signature functions that starts with 'i'

for csv, csvz file formats, file handles will be left open. for xls, ods file formats, the file is read all into memory and is close afterwards. for xlsx, file handles will be left open in python 2.7 - 3.5 by pyexcel-xlsx(openpyxl). In other words, pyexcel-xls, pyexcel-ods, pyexcel-ods3 won't leak file handles.

Saving data to excel file

save_as(**keywords)	Save a sheet from a data source to another one
isave_as(**keywords)	Save a sheet from a data source to another one with less memory
save_book_as(**keywords)	Save a book from a data source to another one
isave_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 source to another one

It accepts two sets of keywords. Why two sets? one set is source, the other set is destination. In order to distinguish 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.

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

Examples on start_row, start_column

Let's assume the following file is a huge csv file:

```
>>> import datetime
>>> import pyexcel as pe
>>> data = [
...     [1, 21, 31],
...     [2, 22, 32],
...     [3, 23, 33],
...     [4, 24, 34],
...     [5, 25, 35],
...     [6, 26, 36]
... ]
>>> pe.save_as(array=data, dest_file_name="your_file.csv")
```

And let's pretend to read partial data:

```
>>> pe.get_sheet(file_name="your_file.csv", start_row=2, row_limit=3)
your_file.csv:
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
```

And you could as well do the same for columns:

```
>>> pe.get_sheet(file_name="your_file.csv", start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 21 | 31 |
+----+----+
| 22 | 32 |
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
| 26 | 36 |
+----+----+
```

Obvious, you could do both at the same time:

```
>>> pe.get_sheet(file_name="your_file.csv",
...     start_row=2, row_limit=3,
...     start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
```

(continues on next page)

(continued from previous page)

```
| 25 | 35 |
+----+----+
```

The pagination support is available across all pyexcel plugins.

> **Note**
>
> No column pagination support for query sets as data source.

Formatting while transcoding a big data file

If you are transcoding a big data set, conventional formatting method would not help unless a on-demand free RAM is available. However, there is a way to minimize the memory footprint of pyexcel while the formatting is performed.

Let's continue from previous example. Suppose we want to transcode "your_file.csv" to "your_file.xls" but increase each element by 1.

What we can do is to define a row renderer function as the following:

```
>>> def increment_by_one(row):
...     for element in row:
...         yield element + 1
```

Then pass it onto save_as function using row_renderer:

```
>>> pe.isave_as(file_name="your_file.csv",
...             row_renderer=increment_by_one,
...             dest_file_name="your_file.xlsx")
```

> **Note**
>
> If the data content is from a generator, isave_as has to be used.

We can verify if it was done correctly:

```
>>> pe.get_sheet(file_name="your_file.xlsx")
your_file.csv:
+---+----+----+
| 2 | 22 | 32 |
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
| 6 | 26 | 36 |
+---+----+----+
| 7 | 27 | 37 |
+---+----+----+
```

Not all parameters are needed. Here is a table

source	parameters
loading from file	file_name, sheet_name, keywords
loading from string	file_content, file_type, sheet_name, keywords
loading from stream	file_stream, file_type, 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
loading from an url	url

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 *file_content* or *file_stream*

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

sheet_name :
sheet name. if sheet_name is not given, the default sheet at index 0 is loaded

start_row
[int] defaults to 0. It allows you to skip rows at the beginning

row_limit: int
defaults to -1, meaning till the end of the whole sheet. It allows you to skip the tailing rows.

start_column
[int] defaults to 0. It allows you to skip columns on your left hand side

column_limit: int
defaults to -1, meaning till the end of the columns. It allows you to skip the tailing columns.

skip_row_func:
It allows you to write your own row skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_column_func:
It allows you to write your own column skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_empty_rows: bool
Defaults to False. Toggle it to True if the rest of empty rows are useless, but it does affect the number of rows.

row_renderer:
You could choose to write a custom row renderer when the data is being read.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially,

encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

Parameters related to xls file format:
Please note the following parameters apply to pyexcel-xls. more details can be found in `xlrd.open_workbook()`

logfile:
An open file to which messages and diagnostics are written.

verbosity:
Increases the volume of trace material written to the logfile.

use_mmap:
Whether to use the mmap module is determined heuristically. Use this arg to override the result.

Current heuristic: mmap is used if it exists.

encoding_override:
Used to overcome missing or bad codepage information in older-version files.

formatting_info:
The default is False, which saves memory.

When True, formatting information will be read from the spreadsheet file. This provides all cells, including empty and blank cells. Formatting information is available for each cell.

ragged_rows:
The default of False means all rows are padded out with empty cells so that all rows have the same size as found in ncols.

True means that there are no empty cells at the ends of rows. This can result in substantial memory savings if rows are of widely varying sizes. See also the row_len() method.

dest_file_name:
another file name.

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

dest_session:
the target database session

dest_table:
the target destination table

dest_model:
the target django model

dest_mapdict:
a mapping dictionary see *pyexcel.Sheet.save_to_memory()*

dest_initializer:
a custom initializer function for table or model

dest_mapdict:
nominate headers

dest_batch_size:
object creation batch size. it is Django specific

dest_library:
choose a specific pyexcel-io plugin for writing

dest_source_library:
choose a specific data source plugin for writing

dest_renderer_library:
choose a pyexcel parser plugin for writing

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

for example: dest_lineterminator will replace default ' ' to the one you specified

In addition, this function use *pyexcel.Sheet* to render the data which could have performance penalty. In exchange, parameters for *pyexcel.Sheet* can be passed on, e.g. *name_columns_by_row*.

pyexcel.isave_as

pyexcel.**isave_as**(***keywords*)

Save a sheet from a data source to another one with less memory

It is similar to *pyexcel.save_as()* except that it does not accept parameters for *pyexcel.Sheet*. And it read when it writes.

It accepts two sets of keywords. Why two sets? one set is source, the other set is destination. In order to distinguish 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.

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

Examples on start_row, start_column

Let's assume the following file is a huge csv file:

```
>>> import datetime
>>> import pyexcel as pe
>>> data = [
...     [1, 21, 31],
...     [2, 22, 32],
```

(continues on next page)

(continued from previous page)

```
...     [3, 23, 33],
...     [4, 24, 34],
...     [5, 25, 35],
...     [6, 26, 36]
... ]
>>> pe.save_as(array=data, dest_file_name="your_file.csv")
```

And let's pretend to read partial data:

```
>>> pe.get_sheet(file_name="your_file.csv", start_row=2, row_limit=3)
your_file.csv:
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
```

And you could as well do the same for columns:

```
>>> pe.get_sheet(file_name="your_file.csv", start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 21 | 31 |
+----+----+
| 22 | 32 |
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
| 26 | 36 |
+----+----+
```

Obvious, you could do both at the same time:

```
>>> pe.get_sheet(file_name="your_file.csv",
...     start_row=2, row_limit=3,
...     start_column=1, column_limit=2)
your_file.csv:
+----+----+
| 23 | 33 |
+----+----+
| 24 | 34 |
+----+----+
| 25 | 35 |
+----+----+
```

The pagination support is available across all pyexcel plugins.

> **Note**
>
> No column pagination support for query sets as data source.

Formatting while transcoding a big data file

If you are transcoding a big data set, conventional formatting method would not help unless a on-demand free RAM is available. However, there is a way to minimize the memory footprint of pyexcel while the formatting is performed.

Let's continue from previous example. Suppose we want to transcode "your_file.csv" to "your_file.xls" but increase each element by 1.

What we can do is to define a row renderer function as the following:

```
>>> def increment_by_one(row):
...     for element in row:
...         yield element + 1
```

Then pass it onto save_as function using row_renderer:

```
>>> pe.isave_as(file_name="your_file.csv",
...             row_renderer=increment_by_one,
...             dest_file_name="your_file.xlsx")
```

> **Note**
>
> If the data content is from a generator, isave_as has to be used.

We can verify if it was done correctly:

```
>>> pe.get_sheet(file_name="your_file.xlsx")
your_file.csv:
+---+----+----+
| 2 | 22 | 32 |
+---+----+----+
| 3 | 23 | 33 |
+---+----+----+
| 4 | 24 | 34 |
+---+----+----+
| 5 | 25 | 35 |
+---+----+----+
| 6 | 26 | 36 |
+---+----+----+
| 7 | 27 | 37 |
+---+----+----+
```

Not all parameters are needed. Here is a table

source	parameters
loading from file	file_name, sheet_name, keywords
loading from string	file_content, file_type, sheet_name, keywords
loading from stream	file_stream, file_type, 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
loading from an url	url

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 *file_content* or *file_stream*

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

sheet_name :
sheet name. if sheet_name is not given, the default sheet at index 0 is loaded

start_row
[int] defaults to 0. It allows you to skip rows at the beginning

row_limit: int
defaults to -1, meaning till the end of the whole sheet. It allows you to skip the tailing rows.

start_column
[int] defaults to 0. It allows you to skip columns on your left hand side

column_limit: int
defaults to -1, meaning till the end of the columns. It allows you to skip the tailing columns.

skip_row_func:
It allows you to write your own row skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_column_func:
It allows you to write your own column skipping functions.

The protocol is to return pyexcel_io.constants.SKIP_DATA if skipping data, pyexcel_io.constants.TAKE_DATA to read data, pyexcel_io.constants.STOP_ITERATION to exit the reading procedure

skip_empty_rows: bool
Defaults to False. Toggle it to True if the rest of empty rows are useless, but it does affect the number of rows.

row_renderer:
You could choose to write a custom row renderer when the data is being read.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially,

encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

Parameters related to xls file format:
Please note the following parameters apply to pyexcel-xls. more details can be found in `xlrd.open_workbook()`

logfile:
An open file to which messages and diagnostics are written.

verbosity:
Increases the volume of trace material written to the logfile.

use_mmap:
Whether to use the mmap module is determined heuristically. Use this arg to override the result.

Current heuristic: mmap is used if it exists.

encoding_override:
Used to overcome missing or bad codepage information in older-version files.

formatting_info:
The default is False, which saves memory.

When True, formatting information will be read from the spreadsheet file. This provides all cells, including empty and blank cells. Formatting information is available for each cell.

ragged_rows:
The default of False means all rows are padded out with empty cells so that all rows have the same size as found in ncols.

True means that there are no empty cells at the ends of rows. This can result in substantial memory savings if rows are of widely varying sizes. See also the row_len() method.

dest_file_name:
another file name.

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

dest_session:
the target database session

dest_table:
the target destination table

dest_model:
the target django model

dest_mapdict:
a mapping dictionary see *pyexcel.Sheet.save_to_memory()*

dest_initializer:
a custom initializer function for table or model

dest_mapdict:
nominate headers

dest_batch_size:
object creation batch size. it is Django specific

dest_library:
choose a specific pyexcel-io plugin for writing

dest_source_library:
choose a specific data source plugin for writing

dest_renderer_library:
choose a pyexcel parser plugin for writing

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

for example: dest_lineterminator will replace default ' ' to the one you specified

In addition, this function use *pyexcel.Sheet* to render the data which could have performance penalty. In exchange, parameters for *pyexcel.Sheet* can be passed on, e.g. *name_columns_by_row*.

When you use this function to work on physical files, this function will leave its file handle open. When you finish the operation on its data, you need to call *pyexcel.free_resources()* to close file handle(s).

for csv, csvz file formats, file handles will be left open. for xls, ods file formats, the file is read all into memory and is close afterwards. for xlsx, file handles will be left open in python 2.7 - 3.5 by pyexcel-xlsx(openpyxl). In other words, pyexcel-xls, pyexcel-ods, pyexcel-ods3 won't leak file handles.

pyexcel.save_book_as

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

Save a book from a data source to another one

Here is a table of parameters:

source	parameters
loading from file	file_name, keywords
loading from string	file_content, file_type, keywords
loading from stream	file_stream, file_type, keywords
loading from sql	session, tables
loading from django models	models
loading from dictionary	bookdict
loading from an url	url

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

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 *file_content* or *file_stream*

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

sheets:
a list of mixed sheet names and sheet indices to be read. This is done to keep Pandas compactibility. With this parameter, more than one sheet can be read and you have the control to read the sheets of your interest instead of all available sheets.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

dest_file_name:
another file name.

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 functions

dest_mapdicts :
to nominate a model or table fields. Optional

dest_batch_size :
batch creation size. Optional

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

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

pyexcel.isave_book_as

pyexcel.**isave_book_as**(***keywords*)

Save a book from a data source to another one

It is similar to *pyexcel.save_book_as()* but it read when it writes. This function provide some speedup but the output data is not made uniform.

Here is a table of parameters:

source	parameters
loading from file	file_name, keywords
loading from string	file_content, file_type, keywords
loading from stream	file_stream, file_type, keywords
loading from sql	session, tables
loading from django models	models
loading from dictionary	bookdict
loading from an url	url

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

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 *file_content* or *file_stream*

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

sheets:
a list of mixed sheet names and sheet indices to be read. This is done to keep Pandas compactibility. With this parameter, more than one sheet can be read and you have the control to read the sheets of your interest instead of all available sheets.

auto_detect_float :
defaults to True

auto_detect_int :
defaults to True

auto_detect_datetime :
defaults to True

ignore_infinity :
defaults to True

library :
choose a specific pyexcel-io plugin for reading

source_library :
choose a specific data source plugin for reading

parser_library :
choose a pyexcel parser plugin for reading

skip_hidden_sheets:
default is True. Please toggle it to read hidden sheets

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

dest_file_name:
another file name.

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 functions

dest_mapdicts :
to nominate a model or table fields. Optional

dest_batch_size :
batch creation size. Optional

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

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

When you use this function to work on physical files, this function will leave its file handle open. When you finish the operation on its data, you need to call *pyexcel.free_resources()* to close file handle(s).

for csv, csvz file formats, file handles will be left open. for xls, ods file formats, the file is read all into memory and is close afterwards. for xlsx, file handles will be left open in python 2.7 - 3.5 by pyexcel-xlsx(openpyxl). In other words, pyexcel-xls, pyexcel-ods, pyexcel-ods3 won't leak file handles.

These flags can be passed on all signature functions:

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.

It is better to uninstall the unwanted pyexcel plugin using pip if two plugins for the same file type are not absolutely necessary.

Cookbook

merge_csv_to_a_book(filelist[, out_file_name])	merge a list of csv files into a excel book
merge_all_to_a_book(filelist[, out_file_name])	merge a list of excel files into a excel book
split_a_book(file_name[, out_file_name])	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*, *out_file_name='merged.xls'*)

merge a list of csv files into a excel book

Parameters

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

pyexcel.merge_all_to_a_book

pyexcel.**merge_all_to_a_book**(*filelist*, *out_file_name='merged.xls'*)

merge a list of excel files into a excel book

Parameters

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

pyexcel.split_a_book

pyexcel.**split_a_book**(*file_name*, *out_file_name=None*)

Split a file into separate sheets

Parameters

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

pyexcel.extract_a_sheet_from_a_book

pyexcel.**extract_a_sheet_from_a_book**(*file_name*, *sheetname*, *out_file_name=None*)

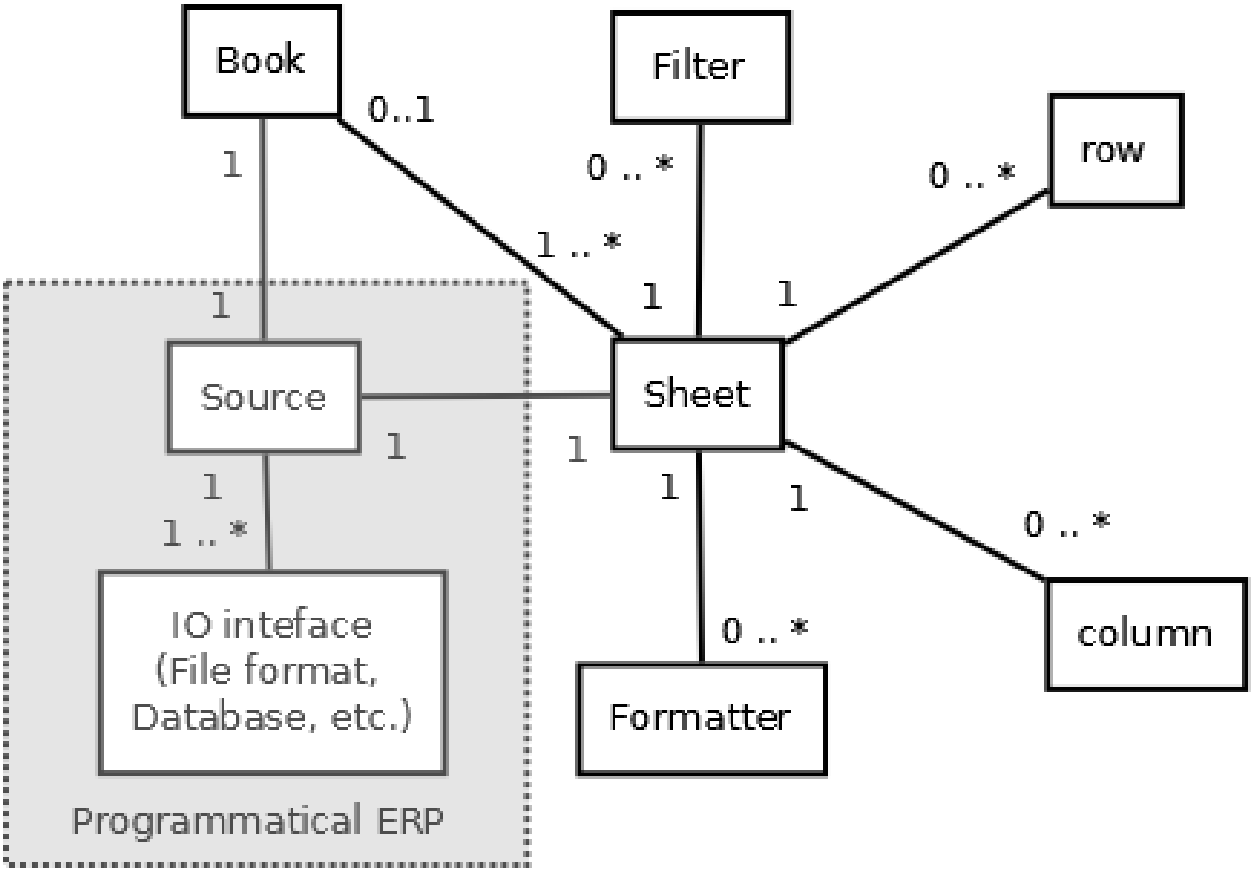
Extract a sheet from a excel book

Parameters

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

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

Parameters

- **sheets** – a dictionary of data
- **filename** – the physical file
- **path** – the relative path or absolute path
- **keywords** – additional parameters to be passed on

Methods

__init__([sheets, filename, path])	Book constructor
get_array(**keywords)	Get data in array format
get_bookdict(**keywords)	Get data in bookdict format
get_csv(**keywords)	Get data in csv format
get_csvz(**keywords)	Get data in csvz format
get_dict(**keywords)	Get data in dict format
get_fods(**__)	fods getter is not defined.
get_handsontable(**keywords)	Get data in handsontable format
get_handsontable_html(**keywords)	Get data in handsontable.html format
get_html(**__)	html getter is not defined.
get_ods(**keywords)	Get data in ods format
get_pdf(**__)	pdf getter is not defined.
get_queryset(**__)	queryset getter is not defined.
get_records(**keywords)	Get data in records format
get_svg(**keywords)	Get data in svg format
get_texttable(**keywords)	Get data in texttable format
get_tsv(**keywords)	Get data in tsv format
get_tsvz(**keywords)	Get data in tsvz format
get_url(**__)	url getter is not defined.
get_xls(**keywords)	Get data in xls format
get_xlsb(**__)	xlsb getter is not defined.
get_xlsm(**keywords)	Get data in xlsm format
get_xlsx(**keywords)	Get data in xlsx format
init([sheets, filename, path])	independent function so that it could be called multiple times
load_from_sheets(sheets)	Load content from existing sheets
number_of_sheets()	Return the number of sheets
plot([file_type])	Visualize the data
register_input(file_type, *[, ...])	create custom attributes for each class
register_io(file_type, *[, instance_name, ...])	create custom attributes for each class
register_presentation(file_type, *[, ...])	create custom attributes for each class
remove_sheet(sheet)	Remove a sheet
save_as(filename, **keywords)	Save the content to a new file
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
set_array(content, **keywords)	Set data in array format
set_bookdict(content, **keywords)	Set data in bookdict format
set_csv(content, **keywords)	Set data in csv format
set_csvz(content, **keywords)	Set data in csvz format
set_dict(content, **keywords)	Set data in dict format
set_fods(content, **keywords)	Set data in fods format
set_handsontable(_y, **_z)	handsontable setter is not defined.
set_handsontable_html(_y, **_z)	handsontable.html setter is not defined.
set_html(content, **keywords)	Set data in html format
set_ods(content, **keywords)	Set data in ods format
set_pdf(content, **keywords)	Set data in pdf format
set_queryset(content, **keywords)	Set data in queryset format
set_records(content, **keywords)	Set data in records format
set_svg(_y, **_z)	svg setter is not defined.

continues on next page

Table 13 – continued from previous page

set_texttable(_y, **_z)	texttable setter is not defined.
set_tsv(content, **keywords)	Set data in tsv format
set_tsvz(content, **keywords)	Set data in tsvz format
set_url(content, **keywords)	Set data in url format
set_xls(content, **keywords)	Set data in xls format
set_xlsb(content, **keywords)	Set data in xlsb format
set_xlsm(content, **keywords)	Set data in xlsm format
set_xlsx(content, **keywords)	Set data in xlsx format
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
sort_sheets([key, reverse])	
to_dict()	Convert the book to a dictionary

Attributes

array	Get/Set data in/from array format
bookdict	Get/Set data in/from bookdict format
csv	Get/Set data in/from csv format
csvz	Get/Set data in/from csvz format
dict	Get/Set data in/from dict format
fods	Set data in fods format
handsontable	Get data in handsontable format
handsontable_html	Get data in handsontable.html format
html	Set data in html format
ods	Get/Set data in/from ods format
pdf	Set data in pdf format
queryset	Set data in queryset format
records	Get/Set data in/from records format
stream	Return a stream in which the content is properly encoded
svg	Get data in svg format
texttable	Get data in texttable format
tsv	Get/Set data in/from tsv format
tsvz	Get/Set data in/from tsvz format
url	Set data in url format
xls	Get/Set data in/from xls format
xlsb	Set data in xlsb format
xlsm	Get/Set data in/from xlsm format
xlsx	Get/Set data in/from xlsx format

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.bookdict	Get/Set data in/from bookdict format
Book.url	Set data in url format
Book.csv	Get/Set data in/from csv format
Book.tsv	Get/Set data in/from tsv format
Book.csvz	Get/Set data in/from csvz format
Book.tsvz	Get/Set data in/from tsvz format
Book.xls	Get/Set data in/from xls format
Book.xlsm	Get/Set data in/from xlsm format
Book.xlsx	Get/Set data in/from xlsx format
Book.ods	Get/Set data in/from ods format
Book.stream	Return a stream in which the content is properly encoded

pyexcel.Book.bookdict

property Book.**bookdict**

Get/Set data in/from bookdict format

You could obtain content in bookdict format by dot notation:

```
Book.bookdict
```

And you could as well set content by dot notation:

```
Book.bookdict = the_io_stream_in_bookdict_format
```

if you need to pass on more parameters, you could use:

```
Book.get_bookdict(**keywords)
Book.set_bookdict(the_io_stream_in_bookdict_format, **keywords)
```

pyexcel.Book.url

property Book.**url**

Set data in url format

You could set content in url format by dot notation:

```
Book.url
```

if you need to pass on more parameters, you could use:

```
Book.set_url(the_io_stream_in_url_format, **keywords)
```

pyexcel.Book.csv

property Book.**csv**

Get/Set data in/from csv format

You could obtain content in csv format by dot notation:

```
Book.csv
```

And you could as well set content by dot notation:

```
Book.csv = the_io_stream_in_csv_format
```

if you need to pass on more parameters, you could use:

```
Book.get_csv(**keywords)
Book.set_csv(the_io_stream_in_csv_format, **keywords)
```

pyexcel.Book.tsv

property Book.**tsv**

Get/Set data in/from tsv format

You could obtain content in tsv format by dot notation:

```
Book.tsv
```

And you could as well set content by dot notation:

```
Book.tsv = the_io_stream_in_tsv_format
```

if you need to pass on more parameters, you could use:

```
Book.get_tsv(**keywords)
Book.set_tsv(the_io_stream_in_tsv_format, **keywords)
```

pyexcel.Book.csvz

property Book.**csvz**

Get/Set data in/from csvz format

You could obtain content in csvz format by dot notation:

```
Book.csvz
```

And you could as well set content by dot notation:

```
Book.csvz = the_io_stream_in_csvz_format
```

if you need to pass on more parameters, you could use:

```
Book.get_csvz(**keywords)
Book.set_csvz(the_io_stream_in_csvz_format, **keywords)
```

pyexcel.Book.tsvz

property Book.**tsvz**

Get/Set data in/from tsvz format

You could obtain content in tsvz format by dot notation:

```
Book.tsvz
```

And you could as well set content by dot notation:

```
Book.tsvz = the_io_stream_in_tsvz_format
```

if you need to pass on more parameters, you could use:

```
Book.get_tsvz(**keywords)
Book.set_tsvz(the_io_stream_in_tsvz_format, **keywords)
```

pyexcel.Book.xls

property Book.**xls**

Get/Set data in/from xls format

You could obtain content in xls format by dot notation:

```
Book.xls
```

And you could as well set content by dot notation:

```
Book.xls = the_io_stream_in_xls_format
```

if you need to pass on more parameters, you could use:

```
Book.get_xls(**keywords)
Book.set_xls(the_io_stream_in_xls_format, **keywords)
```

pyexcel.Book.xlsm

property Book.**xlsm**

Get/Set data in/from xlsm format

You could obtain content in xlsm format by dot notation:

```
Book.xlsm
```

And you could as well set content by dot notation:

```
Book.xlsm = the_io_stream_in_xlsm_format
```

if you need to pass on more parameters, you could use:

```
Book.get_xlsm(**keywords)
Book.set_xlsm(the_io_stream_in_xlsm_format, **keywords)
```

pyexcel.Book.xlsx

property Book.**xlsx**

Get/Set data in/from xlsx format

You could obtain content in xlsx format by dot notation:

```
Book.xlsx
```

And you could as well set content by dot notation:

```
Book.xlsx = the_io_stream_in_xlsx_format
```

if you need to pass on more parameters, you could use:

```
Book.get_xlsx(**keywords)
Book.set_xlsx(the_io_stream_in_xlsx_format, **keywords)
```

pyexcel.Book.ods

property Book.**ods**

Get/Set data in/from ods format

You could obtain content in ods format by dot notation:

```
Book.ods
```

And you could as well set content by dot notation:

```
Book.ods = the_io_stream_in_ods_format
```

if you need to pass on more parameters, you could use:

```
Book.get_ods(**keywords)
Book.set_ods(the_io_stream_in_ods_format, **keywords)
```

pyexcel.Book.stream

property Book.**stream**

Return a stream in which the content is properly encoded

Example:

```
>>> import pyexcel as p
>>> b = p.get_book(bookdict={"A": [[1]]})
>>> csv_stream = b.stream.texttable
>>> print(csv_stream.getvalue())
A:
+---+
| 1 |
+---+
```

Where b.stream.xls.getvalue() is equivalent to b.xls. In some situation b.stream.xls is preferred than b.xls.

Sheet examples:

```
>>> import pyexcel as p
>>> s = p.Sheet([[1]], 'A')
>>> csv_stream = s.stream.texttable
>>> print(csv_stream.getvalue())
A:
+---+
| 1 |
+---+
```

Where s.stream.xls.getvalue() is equivalent to s.xls. In some situation s.stream.xls is preferred than s.xls.

It is similar to *save_to_memory()*.

Save changes

Book.save_as(filename, **keywords)	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
Book.save_to_django_models(models[, ...])	Save to database table through django model

pyexcel.Book.save_as

Book.**save_as**(*filename*, ***keywords*)

Save the content to a new file

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

PARAMETERS

filename: a file path

library:
choose a specific pyexcel-io plugin for writing

renderer_library:
choose a pyexcel parser plugin for writing

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

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 initialization 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.

pyexcel.Book.save_to_django_models

Book.**save_to_django_models**(*models*, *initializers=None*, *mapdicts=None*, ***keywords*)

Save to database table through django model

Parameters

- **models** – a list of database models, that is accepted by *Sheet.save_to_django_model()*. 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 initialization functions for your tables and the sequence should match tables,

- **mapdicts** – custom map dictionary for your data columns and the sequence should match tables

optional parameters: :param batch_size: django bulk_create batch size :param bulk_save: whether to use bulk_create or to use single save

per record

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 these 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.

Variables

- **name** – sheet name. use to change sheet name
- **row** – access data row by row
- **column** – access data column by column

Example:

```
>>> import pyexcel as p
>>> content = {'A': [[1]]}
>>> b = p.get_book(bookdict=content)
>>> b
A:
+---+
| 1 |
+---+
>>> b[0].name
'A'
>>> b[0].name = 'B'
>>> b
B:
+---+
```

(continues on next page)

(continued from previous page)

```
| 1 |
+---+
```

__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
cell_value(row, column[, new_value])	Random access to table cells
clone()	
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([column_indices, row_indices])	Apply the filter with immediate effect
format(formatter)	Apply a formatting action for the whole sheet
get_array(**keywords)	Get data in array format
get_bookdict(**keywords)	Get data in bookdict format
get_csv(**keywords)	Get data in csv format
get_csvz(**keywords)	Get data in csvz format
get_dict(**keywords)	Get data in dict format
get_fods(**__)	fods getter is not defined.
get_handsontable(**keywords)	Get data in handsontable format
get_handsontable_html(**keywords)	Get data in handsontable.html format
get_html(**__)	html getter is not defined.
get_internal_array()	present internal array

continues on next page

Table 19 – continued from previous page

get_ods(**keywords)	Get data in ods format
get_pdf(**__)	pdf getter is not defined.
get_queryset(**__)	queryset getter is not defined.
get_records(**keywords)	Get data in records format
get_svg(**keywords)	Get data in svg format
get_texttable(**keywords)	Get data in texttable format
get_tsv(**keywords)	Get data in tsv format
get_tsvz(**keywords)	Get data in tsvz format
get_url(**__)	url getter is not defined.
get_xls(**keywords)	Get data in xls format
get_xlsb(**__)	xlsb getter is not defined.
get_xlsm(**keywords)	Get data in xlsm format
get_xlsx(**keywords)	Get data in xlsx format
group_rows_by_column(column_index_or_name)	Group rows with similar column into a two-dimensional array.
init([sheet, name, name_columns_by_row, ...])	custom initialization functions
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()	iterate rows using column names
named_row_at(name)	Get a row by its name
named_rows()	iterate rows using row names
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
plot([file_type])	Visualize the data
project(new_ordered_columns[, exclusion])	Rearrange the sheet.
rcolumns()	Returns a right to left column iterator
region(topleft_corner, bottomright_corner)	Get a rectangle shaped data out
register_input(file_type[, instance_name, ...])	create custom attributes for each class
register_io(file_type[, instance_name, ...])	create custom attributes for each class
register_presentation(file_type[, ...])	create custom attributes for each class
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 column to leftmost row and from bottom to top example.
save_as(filename, **keywords)	Save the content to a named file
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_array(content, **keywords)	Set data in array format
set_bookdict(content, **keywords)	Set data in bookdict format
set_column_at(column_index, data_array[, ...])	Updates a column data range
set_csv(content, **keywords)	Set data in csv format
set_csvz(content, **keywords)	Set data in csvz format

continues on next page

Table 19 – continued from previous page

set_dict(content, **keywords)	Set data in dict format
set_fods(content, **keywords)	Set data in fods format
set_handsontable(_y, **_z)	handsontable setter is not defined.
set_handsontable_html(_y, **_z)	handsontable.html setter is not defined.
set_html(content, **keywords)	Set data in html format
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_ods(content, **keywords)	Set data in ods format
set_pdf(content, **keywords)	Set data in pdf format
set_queryset(content, **keywords)	Set data in queryset format
set_records(content, **keywords)	Set data in records format
set_row_at(row_index, data_array)	Update a row data range
set_svg(_y, **_z)	svg setter is not defined.
set_texttable(_y, **_z)	texttable setter is not defined.
set_tsv(content, **keywords)	Set data in tsv format
set_tsvz(content, **keywords)	Set data in tsvz format
set_url(content, **keywords)	Set data in url format
set_xls(content, **keywords)	Set data in xls format
set_xlsb(content, **keywords)	Set data in xlsb format
set_xlsm(content, **keywords)	Set data in xlsm format
set_xlsx(content, **keywords)	Set data in xlsx format
to_array()	Returns an array after filtering
to_dict([row])	Returns a dictionary
to_records([custom_headers])	Make an array of dictionaries
top([lines])	Preview top most 5 rows
top_left([rows, columns])	Preview top corner: 5x5
transpose()	Rotate the data table by 90 degrees
vertical()	Default iterator to go through each cell one by one from leftmost column to rightmost row and from top to bottom example.

Attributes

array	Get/Set data in/from array format
bookdict	Get/Set data in/from bookdict format
colnames	Return column names if any
content	Plain representation without headers
csv	Get/Set data in/from csv format
csvz	Get/Set data in/from csvz format
dict	Get/Set data in/from dict format
fods	Set data in fods format
handsontable	Get data in handsontable format
handsontable_html	Get data in handsontable.html format
html	Set data in html format
ods	Get/Set data in/from ods format
pdf	Set data in pdf format
queryset	Set data in queryset format
records	Get/Set data in/from records format
rownames	Return row names if any
stream	Return a stream in which the content is properly encoded

continues on next page

Table 20 – continued from previous page

svg	Get data in svg format
texttable	Get data in texttable format
tsv	Get/Set data in/from tsv format
tsvz	Get/Set data in/from tsvz format
url	Set data in url format
xls	Get/Set data in/from xls format
xlsb	Set data in xlsb format
xlsm	Get/Set data in/from xlsm format
xlsx	Get/Set data in/from xlsx format

Attributes

Sheet.content	Plain representation without headers
Sheet.number_of_rows()	The number of rows
Sheet.number_of_columns()	The number of columns
Sheet.row_range()	Utility function to get row range
Sheet.column_range()	Utility function to get column range

pyexcel.Sheet.content

property Sheet.**content**

Plain representation without headers

pyexcel.Sheet.number_of_rows

Sheet.**number_of_rows**()

The number of rows

pyexcel.Sheet.number_of_columns

Sheet.**number_of_columns**()

The number of columns

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

Cell access

Sheet.cell_value(row, column[, new_value])	Random access to table cells

continues on next page

Table 22 – continued from previous page

Sheet.__getitem__(aset)	By default, this class recognize from top to bottom from left to right

pyexcel.Sheet.cell_value

Sheet.**cell_value**(*row*, *column*, *new_value=None*)

Random access to table cells

Parameters

- **row** (*int*) – row index which starts from 0
- **column** (*int*) – column index which starts from 0
- **new_value** (*any*) – new value if this is to set the value

pyexcel.Sheet.__getitem__

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

By default, this class recognize from top to bottom from left to right

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*)

Update a row data 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 if any
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 row_index: the index of the row that has the column names

pyexcel.Sheet.rownames

property Sheet.**rownames**

Return row names if any

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 if any
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

continues on next page

Table 26 – continued from previous page

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 column_index: the index of the column that has the row names

pyexcel.Sheet.colnames

property Sheet.**colnames**

Return column names if any

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.

Conversion

Sheet.array	Get/Set data in/from array format
Sheet.records	Get/Set data in/from records format
Sheet.dict	Get/Set data in/from dict format
Sheet.url	Set data in url format
Sheet.csv	Get/Set data in/from csv format
Sheet.tsv	Get/Set data in/from tsv format
Sheet.csvz	Get/Set data in/from csvz format
Sheet.tsvz	Get/Set data in/from tsvz format
Sheet.xls	Get/Set data in/from xls format
Sheet.xlsm	Get/Set data in/from xlsm format
Sheet.xlsx	Get/Set data in/from xlsx format
Sheet.ods	Get/Set data in/from ods format
Sheet.stream	Return a stream in which the content is properly encoded

pyexcel.Sheet.array

property Sheet.**array**

Get/Set data in/from array format

You could obtain content in array format by dot notation:

```
Sheet.array
```

And you could as well set content by dot notation:

```
Sheet.array = the_io_stream_in_array_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_array(**keywords)
Sheet.set_array(the_io_stream_in_array_format, **keywords)
```

pyexcel.Sheet.records

property Sheet.**records**

Get/Set data in/from records format

You could obtain content in records format by dot notation:

```
Sheet.records
```

And you could as well set content by dot notation:

```
Sheet.records = the_io_stream_in_records_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_records(**keywords)
Sheet.set_records(the_io_stream_in_records_format, **keywords)
```

pyexcel.Sheet.dict

property Sheet.**dict**

Get/Set data in/from dict format

You could obtain content in dict format by dot notation:

```
Sheet.dict
```

And you could as well set content by dot notation:

```
Sheet.dict = the_io_stream_in_dict_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_dict(**keywords)
Sheet.set_dict(the_io_stream_in_dict_format, **keywords)
```

pyexcel.Sheet.url

property Sheet.**url**

Set data in url format

You could set content in url format by dot notation:

```
Sheet.url
```

if you need to pass on more parameters, you could use:

```
Sheet.set_url(the_io_stream_in_url_format, **keywords)
```

pyexcel.Sheet.csv

property Sheet.**csv**

Get/Set data in/from csv format

You could obtain content in csv format by dot notation:

```
Sheet.csv
```

And you could as well set content by dot notation:

```
Sheet.csv = the_io_stream_in_csv_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_csv(**keywords)
Sheet.set_csv(the_io_stream_in_csv_format, **keywords)
```

pyexcel.Sheet.tsv

property Sheet.**tsv**

Get/Set data in/from tsv format

You could obtain content in tsv format by dot notation:

```
Sheet.tsv
```

And you could as well set content by dot notation:

```
Sheet.tsv = the_io_stream_in_tsv_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_tsv(**keywords)
Sheet.set_tsv(the_io_stream_in_tsv_format, **keywords)
```

pyexcel.Sheet.csvz

property Sheet.**csvz**

Get/Set data in/from csvz format

You could obtain content in csvz format by dot notation:

```
Sheet.csvz
```

And you could as well set content by dot notation:

```
Sheet.csvz = the_io_stream_in_csvz_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_csvz(**keywords)
Sheet.set_csvz(the_io_stream_in_csvz_format, **keywords)
```

pyexcel.Sheet.tsvz

property Sheet.**tsvz**

Get/Set data in/from tsvz format

You could obtain content in tsvz format by dot notation:

```
Sheet.tsvz
```

And you could as well set content by dot notation:

```
Sheet.tsvz = the_io_stream_in_tsvz_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_tsvz(**keywords)
Sheet.set_tsvz(the_io_stream_in_tsvz_format, **keywords)
```

pyexcel.Sheet.xls

property Sheet.**xls**

Get/Set data in/from xls format

You could obtain content in xls format by dot notation:

```
Sheet.xls
```

And you could as well set content by dot notation:

```
Sheet.xls = the_io_stream_in_xls_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_xls(**keywords)
Sheet.set_xls(the_io_stream_in_xls_format, **keywords)
```

pyexcel.Sheet.xlsm

property Sheet.**xlsm**

Get/Set data in/from xlsm format

You could obtain content in xlsm format by dot notation:

```
Sheet.xlsm
```

And you could as well set content by dot notation:

```
Sheet.xlsm = the_io_stream_in_xlsm_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_xlsm(**keywords)
Sheet.set_xlsm(the_io_stream_in_xlsm_format, **keywords)
```

pyexcel.Sheet.xlsx

property Sheet.**xlsx**

Get/Set data in/from xlsx format

You could obtain content in xlsx format by dot notation:

```
Sheet.xlsx
```

And you could as well set content by dot notation:

```
Sheet.xlsx = the_io_stream_in_xlsx_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_xlsx(**keywords)
Sheet.set_xlsx(the_io_stream_in_xlsx_format, **keywords)
```

pyexcel.Sheet.ods

property Sheet.**ods**

Get/Set data in/from ods format

You could obtain content in ods format by dot notation:

```
Sheet.ods
```

And you could as well set content by dot notation:

```
Sheet.ods = the_io_stream_in_ods_format
```

if you need to pass on more parameters, you could use:

```
Sheet.get_ods(**keywords)
Sheet.set_ods(the_io_stream_in_ods_format, **keywords)
```

pyexcel.Sheet.stream

property Sheet.**stream**

Return a stream in which the content is properly encoded

Example:

```
>>> import pyexcel as p
>>> b = p.get_book(bookdict={"A": [[1]]})
>>> csv_stream = b.stream.texttable
>>> print(csv_stream.getvalue())
A:
+---+
| 1 |
+---+
```

Where b.stream.xls.getvalue() is equivalent to b.xls. In some situation b.stream.xls is preferred than b.xls.

Sheet examples:

```
>>> import pyexcel as p
>>> s = p.Sheet([[1]], 'A')
>>> csv_stream = s.stream.texttable
>>> print(csv_stream.getvalue())
A:
+---+
| 1 |
+---+
```

Where s.stream.xls.getvalue() is equivalent to s.xls. In some situation s.stream.xls is preferred than s.xls.

It is similar to *save_to_memory()*.

Formatting

Sheet.format(formatter)	Apply a formatting action for the whole sheet

pyexcel.Sheet.format

Sheet.**format**(*formatter*)

Apply a formatting action for the whole sheet

Example:

```
>>> import pyexcel as p
>>> # Given a dictionary 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 = p.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)
```

(continues on next page)

(continued from previous page)

```
>>> sheet.row[1]
[1, 1, 2, 1]
```

Filtering

Sheet.filter([column_indices, row_indices])	Apply the filter with immediate effect

pyexcel.Sheet.filter

Sheet.**filter**(*column_indices=None*, *row_indices=None*)

Apply the filter with immediate effect

Transformation

Sheet.project(new_ordered_columns[, exclusion])	Rearrange the sheet.
Sheet.transpose()	Rotate 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.project

Sheet.**project**(*new_ordered_columns*, *exclusion=False*)

Rearrange the sheet.

Variables

- **new_ordered_columns** – new columns
- **exclusion** – to exclude named column or not. defaults to False

Example:

```
>>> sheet = Sheet(
... [["A", "B", "C"], [1, 2, 3], [11, 22, 33], [111, 222, 333]],
... name_columns_by_row=0)
>>> sheet.project(["B", "A", "C"])
pyexcel sheet:
+-----+-----+-----+
|  B  |  A  |  C  |
+=====+=====+=====+
| 2   | 1   | 3   |
+-----+-----+-----+
| 22  | 11  | 33  |
+-----+-----+-----+
| 222 | 111 | 333 |
+-----+-----+-----+
>>> sheet.project(["B", "C"])
```

(continues on next page)

(continued from previous page)

```
pyexcel sheet:
+-----+-----+
|  B  |  C  |
+=====+=====+
| 2   | 3   |
+-----+-----+
| 22  | 33  |
+-----+-----+
| 222 | 333 |
+-----+-----+
>>> sheet.project(["B", "C"], exclusion=True)
pyexcel sheet:
+-----+
|  A  |
+=====+
| 1   |
+-----+
| 11  |
+-----+
| 111 |
+-----+
```

pyexcel.Sheet.transpose

Sheet.**transpose**()

Rotate 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 p
>>> # Given a dictionary 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 = p.get_sheet(adict=data)
>>> sheet.row[1]
[1, 1.25, 2, 1]
>>> inc = lambda value: (float(value) if value != '' 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

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

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 p
>>> 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 = p.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 |
```

(continues on next page)

(continued from previous page)

```
+----+----+----+----+----+----+----+----+----+----+
|    |    |    |    |    |    | 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 |
+----+----+----+----+----+----+----+----+----+----+----+----+
```

Save changes

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
Sheet.save_to_django_model(model[, ...])	Save to database table through django model

pyexcel.Sheet.save_as

Sheet.**save_as**(*filename*, ***keywords*)

Save the content to a named file

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

PARAMETERS

filename: a file path

library:
choose a specific pyexcel-io plugin for writing

renderer_library:
choose a pyexcel parser plugin for writing

Parameters related to csv file format

for csv, fmtparams are accepted

delimiter :
field separator

lineterminator :
line terminator

encoding:
csv specific. Specify the file encoding the csv file. For example: encoding='latin1'. Especially, encoding='utf-8-sig' would add utf 8 bom header if used in renderer, or would parse a csv with utf brom header used in parser.

escapechar :
A one-character string used by the writer to escape the delimiter if quoting is set to QUOTE_NONE and the quotechar if doublequote is False.

quotechar :
A one-character string used to quote fields containing special characters, such as the delimiter or quotechar, or which contain new-line characters. It defaults to '"'

quoting :
Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the QUOTE_* constants (see section Module Contents) and defaults to QUOTE_MINIMAL.

skipinitialspace :
When True, whitespace immediately following the delimiter is ignored. The default is False.

pep_0515_off :
When True in python version 3.6, PEP-0515 is turned on. The default is False

pyexcel.Sheet.save_to_memory

Sheet.**save_to_memory**(*file_type*, *stream=None*, ***keywords*)

Save the content to memory

Parameters

- **file_type** – any value of 'csv', 'tsv', 'csvz', 'tsvz', 'xls', 'xlsm', 'xlsm', 'ods'
- **stream** – 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 initialization functions for your table
- **mapdict** – custom map dictionary for your data columns
- **auto_commit** – by default, data is auto committed.

pyexcel.Sheet.save_to_django_model

Sheet.**save_to_django_model**(*model*, *initializer=None*, *mapdict=None*, *batch_size=None*)

Save to database table through django model

Parameters

- **model** – a database model
- **initializer** – a initialization functions for your model
- **mapdict** – custom map dictionary for your data columns
- **batch_size** – a parameter to Django concerning the size for bulk insertion

2.9.2 Internal API reference

This is intended for developers and hackers of pyexcel.

Data sheet representation

In inheritance order from parent to child

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

pyexcel.internal.sheets.Matrix

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

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

__init__(*array*)

Parameters

array (*list*) – a list of arrays or a generator

Methods

__init__(array)	
cell_value(row, column[, new_value])	Random access to table cells
clone()	
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 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 dimensional data after the bottom row

continues on next page

Table 33 – continued from previous page

filter([column_indices, row_indices])	Apply the filter with immediate effect
format(formatter)	Apply a formatting action for the whole sheet
get_array(**keywords)	Get data in array format
get_bookdict(**keywords)	Get data in bookdict format
get_csv(**keywords)	Get data in csv format
get_csvz(**keywords)	Get data in csvz format
get_dict(**keywords)	Get data in dict format
get_fods(**__)	fods getter is not defined.
get_handsontable(**keywords)	Get data in handsontable format
get_handsontable_html(**keywords)	Get data in handsontable.html format
get_html(**__)	html getter is not defined.
get_internal_array()	present internal array
get_ods(**keywords)	Get data in ods format
get_pdf(**__)	pdf getter is not defined.
get_queryset(**__)	queryset getter is not defined.
get_records(**keywords)	Get data in records format
get_svg(**keywords)	Get data in svg format
get_texttable(**keywords)	Get data in texttable format
get_tsv(**keywords)	Get data in tsv format
get_tsvz(**keywords)	Get data in tsvz format
get_url(**__)	url getter is not defined.
get_xls(**keywords)	Get data in xls format
get_xlsb(**__)	xlsb getter is not defined.
get_xlsm(**keywords)	Get data in xlsm format
get_xlsx(**keywords)	Get data in xlsx format
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
plot([file_type])	Visualize the data
rcolumns()	Returns a right to left column iterator
region(topleft_corner, bottomright_corner)	Get a rectangle shaped data out
register_input(file_type[, instance_name, ...])	create custom attributes for each class
register_io(file_type[, instance_name, ...])	create custom attributes for each class
register_presentation(file_type[, ...])	create custom attributes for each class
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 column to leftmost row and from bottom to top example.
save_as(filename, **keywords)	Save the content to a named file
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_array(content, **keywords)	Set data in array format
set_bookdict(content, **keywords)	Set data in bookdict format
set_column_at(column_index, data_array[, ...])	Updates a column data range
set_csv(content, **keywords)	Set data in csv format
set_csvz(content, **keywords)	Set data in csvz format

continues on next page

Table 33 – continued from previous page

set_dict(content, **keywords)	Set data in dict format
set_fods(content, **keywords)	Set data in fods format
set_handsontable(_y, **_z)	handsontable setter is not defined.
set_handsontable_html(_y, **_z)	handsontable.html setter is not defined.
set_html(content, **keywords)	Set data in html format
set_ods(content, **keywords)	Set data in ods format
set_pdf(content, **keywords)	Set data in pdf format
set_queryset(content, **keywords)	Set data in queryset format
set_records(content, **keywords)	Set data in records format
set_row_at(row_index, data_array)	Update a row data range
set_svg(_y, **_z)	svg setter is not defined.
set_texttable(_y, **_z)	texttable setter is not defined.
set_tsv(content, **keywords)	Set data in tsv format
set_tsvz(content, **keywords)	Set data in tsvz format
set_url(content, **keywords)	Set data in url format
set_xls(content, **keywords)	Set data in xls format
set_xlsb(content, **keywords)	Set data in xlsb format
set_xlsm(content, **keywords)	Set data in xlsm format
set_xlsx(content, **keywords)	Set data in xlsx format
to_array()	Get an array out
transpose()	Rotate the data table by 90 degrees
vertical()	Default iterator to go through each cell one by one from leftmost column to rightmost row and from top to bottom example.

Attributes

array	Get/Set data in/from array format
bookdict	Get/Set data in/from bookdict format
csv	Get/Set data in/from csv format
csvz	Get/Set data in/from csvz format
dict	Get/Set data in/from dict format
fods	Set data in fods format
handsontable	Get data in handsontable format
handsontable_html	Get data in handsontable.html format
html	Set data in html format
ods	Get/Set data in/from ods format
pdf	Set data in pdf format
queryset	Set data in queryset format
records	Get/Set data in/from records format
stream	Return a stream in which the content is properly encoded
svg	Get data in svg format
texttable	Get data in texttable format
tsv	Get/Set data in/from tsv format
tsvz	Get/Set data in/from tsvz format
url	Set data in url format
xls	Get/Set data in/from xls format
xlsb	Set data in xlsb format
xlsm	Get/Set data in/from xlsm format

continues on next page

Table 34 – continued from previous page

xlsx	Get/Set data in/from xlsx format

SheetStream(name, payload)	Memory efficient sheet representation
BookStream([sheets, filename, path])	Memory efficient book representation

pyexcel.internal.generators.SheetStream

class pyexcel.internal.generators.**SheetStream**(*name*, *payload*)

Memory efficient sheet representation

This class wraps around the data read from pyexcel-io. Comparing with *pyexcel.Sheet*, the instance of this class does not load all data into memory. Hence it performs better when dealing with big data.

If you would like to do custom rendering for each row of the two dimensional data, you would need to pass a row formatting/rendering function to the parameter "renderer" of pyexcel's signature functions.

__init__(*name*, *payload*)

Methods

__init__(name, payload)	
get_internal_array()	
to_array()	Simply return the generator

Attributes

array	array attribute

pyexcel.internal.generators.BookStream

class pyexcel.internal.generators.**BookStream**(*sheets=None*, *filename='memory'*, *path=None*)

Memory efficient book representation

Comparing with *pyexcel.Book*, the instance of this class uses pyexcel.generators.SheetStream as its internal representation of sheet objects. Because *SheetStream* does not read data into memory, it is memory efficient.

__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 absolute path :param set keywords: additional parameters to be passed on

Methods

__init__([sheets, filename, path])	Book constructor
load_from_sheets(sheets)	Load content from existing sheets

continues on next page

Table 38 – continued from previous page

number_of_sheets()	Return the number of sheets
sheet_names()	
to_dict()	Get book data structure as a dictionary

Row representation

Row(matrix)	Represent row of a matrix

pyexcel.internal.sheets.Row

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

Represent row of a matrix

Table 40: "example.csv"

1	2	3
4	5	6
7	8	9

Above column manipulation can be performed on rows similarly. 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.internal.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)	
format([row_index, formatter, format_specs])	Format a row
get_converter(theformatter)	return the actual converter or a built-in converter
select(indices)	Delete row indices other than specified

Column representation

Column(matrix)	Represent columns of a matrix

pyexcel.internal.sheets.Column

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

Represent columns of a matrix

Table 43: "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.internal.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)	
format([column_index, formatter, format_specs])	Format a column
get_converter(theformatter)	return the actual converter or a built-in converter
select(indices)	Examples:

2.10 Developer's guide

2.10.1 Developer's guide

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

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

Once you have finished your changes, please provide test case(s), relevant documentation.

> **Note**
>
> As to rnd_requirements.txt, usually, it is created when a dependent library is not released. Once the dependency is installed (will be released), the future version of the dependency in the requirements.txt will be valid.

How to test your contribution

Although *nose* and *doctest* are both used in code testing, it is advisable 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
```

On Windows, please issue this command:

```
> test.bat
```

Before you commit

Please run:

```
$ make format
```

so as to beautify your code otherwise your build may fail your unit test.

Before you raise pull request

Please edit 'changelog.yml' and record your changes

2.10.2 How to log pyexcel

When developing source plugins, it becomes necessary to have log trace available. It helps find out what goes wrong quickly.

The basic step would be to set up logging before pyexcel import statement.

```
import logging
import logging.config
logging.basicConfig(format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    level=logging.DEBUG)

import pyexcel
```

And if you would use a complex configuration, you can use the following code.

```
import logging
import logging.config
logging.config.fileConfig('log.conf')
```

(continues on next page)

(continued from previous page)

```
import pyexcel
```

And then save the following content as log.conf in your directory:

```
   [loggers]
   keys=root, sources, renderers

   [handlers]
   keys=consoleHandler

   [formatters]
   keys=custom

   [logger_root]
   level=INFO
   handlers=consoleHandler

   [logger_sources]
   level=DEBUG
   handlers=consoleHandler
   qualname=pyexcel.sources.factory
   propagate=0

   [logger_renderers]
   level=DEBUG
   handlers=consoleHandler
   qualname=pyexcel.renderers.factory
   propagate=0

   [handler_consoleHandler]
   class=StreamHandler
   level=DEBUG
   formatter=custom
   args=(sys.stdout,)

   [formatter_custom]
   format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
   datefmt=


Disable logging
-------------------

In unit testing and django framework, you will find the `lml` logging even you have not
explicitly wanted them.


You can suppress them:
```

```
  import logging
  logging.getLogger('lml.plugin').propagate = False
```

(continues on next page)

(continued from previous page)

```
Here is a list of possible modules: `lml.plugin` and `lml.loader`.
```

2.10.3 Packaging with PyInstaller

With pyexcel v0.5.0, the way to package it has been changed because it uses lml for all plugins.

And you need to do the same for pyexcel-io plugins too.

Built-in plugins of pyexcel

In order to package every built-in plugins of pyexcel-io, you need to specify:

```
--hidden-import pyexcel.plugins.renderers.sqlalchemy
--hidden-import pyexcel.plugins.renderers.django
--hidden-import pyexcel.plugins.renderers.excel
--hidden-import pyexcel.plugins.renderers._texttable
--hidden-import pyexcel.plugins.parsers.excel
--hidden-import pyexcel.plugins.parsers.sqlalchemy
--hidden-import pyexcel.plugins.sources.http
--hidden-import pyexcel.plugins.sources.file_input
--hidden-import pyexcel.plugins.sources.memory_input
--hidden-import pyexcel.plugins.sources.file_output
--hidden-import pyexcel.plugins.sources.output_to_memory
--hidden-import pyexcel.plugins.sources.pydata.bookdict
--hidden-import pyexcel.plugins.sources.pydata.dictsource
--hidden-import pyexcel.plugins.sources.pydata.arraysource
--hidden-import pyexcel.plugins.sources.pydata.records
--hidden-import pyexcel.plugins.sources.django
--hidden-import pyexcel.plugins.sources.sqlalchemy
--hidden-import pyexcel.plugins.sources.querysets
```

2.10.4 How to write a plugin for pyexcel

> **Note**
>
> Under writing. Stay tuned.

There are three types of plugins for pyexcel: data parser, data renderer and data source.

Tutorial

Let me walk you through the process of creating pyexcel-pdfr package.

Prerequisites:

1. pip install moban yehua
2. git clone https://github.com/moremoban/setupmobans.git # generic setup
3. git clone https://github.com/pyexcel/pyexcel-commons.git

Let me assume that you have the work directory as:

```
setupmobans pyexcel-commons
```

and *YOUR_WORK_DIRECTORY* points to the base directory for both.

And then please export an environment variable:

```
export YEHUA_FILE=$YOUR_WORK_DIRECTORY/pyexcel-commons/yehua/yehua.yml
```

Now let's get started.

Step 1

Call *yehua* to get the basic scaffolding:

```
$ yehua
Yehua will walk you through creating a pyexcel package.
Press ^C to quit at any time.

What is your project name? pyexcel-pdfr
What is the description? parses tables in pdf file as tabular data
What is project type?
1. pyexcel plugins
2. command line interface
3. python's C externsion
(1,2,3): 1
What is the nick name? pdf
$
```

Step 2

Call *moban* to inflate all project files:

```
$ cd pyexcel-pdfr/
$ ln -s ../pyexcel-commons/ commons
$ ln -s ../setupmobans/ setupmobans
$ moban
Templating README.rst to README.rst
Templating setup.py to setup.py
Templating requirements.txt to requirements.txt
Templating NEW_BSD_LICENSE.jj2 to LICENSE
Templating MANIFEST.in.jj2 to MANIFEST.in
Templating tests/requirements.txt to tests/requirements.txt
Templating test.script.jj2 to test.sh
Templating test.script.jj2 to test.bat
Templating travis.yml.jj2 to .travis.yml
Templating gitignore.jj2 to .gitignore
Templating docs/source/conf.py.jj2 to docs/source/conf.py
```

Step 3 - Coding

Please put your code in pyexcel_pdfr

2.11 Change log

2.11.1 What's breaking in 0.7.0

The following statements will stop working:

```
import pyexcel.ext.ods
import pyexcel.ext.ods3
import pyexcel.ext.text
import pyexcel.ext.xls
import pyexcel.ext.xlsx
```

as they were deprecated since v0.2.2

2.11.2 What's breaking in 0.6.0

In the following statements:

```
sheet_a = sheet.row + rows
sheet_b = sheet.column + columns
book = sheet_a + sheet_b
```

sheet_a and *sheet_b* will no longer have access to the data of *sheet*. *book* will no longer have access to the data of *sheet_a* and *sheet_b*.

Under Hyrum's Law, this enhancement in 0.6.0 will cause breakage otherwise.

2.11.3 What's breaking in 0.5.9

`pyexcel.Sheet.to_records()` returns a generator instead of a list of dictionaries.

2.11.4 Migrate away from 0.4.3

get_{{file_type}}_stream functions from pyexcel.Sheet and pyexcel.Book were introduced since 0.4.3 but were removed since 0.4.4. Please be advised to use save_to_memory functions, Sheet.io.{{file_type}} or Book.io.{{file_type}}.

2.11.5 Migrate from 0.2.x to 0.3.0+

Filtering and formatting behavior of *pyexcel.Sheet* are simplified. Soft filter and soft formatter are removed. Extra classes such as iterator, formatter, filter are removed.

Most of formatting tasks could be achieved using *format()* and *map()*. and Filtering with *filter()*. Formatting and filtering on row and/or column can be found with `row()` and `column()`

1. Updated filter function

There is no alternative to replace the following code:

```
sheet.filter(pe.OddRowFilter())
```

You will need to remove odd rows by yourself:

```
>>> import pyexcel as pe
>>> data = [
...  ['1'],
```

(continues on next page)

(continued from previous page)

```
...  ['2'],
...  ['3'],
... ]
>>> sheet = pe.Sheet(data)
>>> to_remove = []
>>> for index in sheet.row_range():
...     if index % 2 == 0:
...         to_remove.append(index)
>>> sheet.filter(row_indices=to_remove)
>>> sheet
pyexcel sheet:
+---+
| 2 |
+---+
```

Or, you could do this:

```
>>> data = [
...  ['1'],
...  ['2'],
...  ['3'],
... ]
>>> sheet = pe.Sheet(data)
>>> def odd_filter(row_index, _):
...    return row_index % 2 == 0
>>> del sheet.row[odd_filter]
>>> sheet
pyexcel sheet:
+---+
| 2 |
+---+
```

And the same applies to EvenRowFilter, OddColumnFilter, EvenColumnFilter.

2. Updated format function

2.1 Replacement of sheetformatter

The following formatting code:

```
sheet.apply_formatter(pe.sheets.formatters.SheetFormatter(int))
```

can be replaced by:

```
sheet.format(int)
```

2.2 Replacement of row formatters

The following code:

```
row_formatter = pe.sheets.formatters.RowFormatter([1, 2], str)
sheet.add_formatter(row_formatter)
```

can be replaced by:

```
sheet.row.format([1, 2], str)
```

2.3 Replacement of column formatters

The following code:

```
f = NamedColumnFormatter(["Column 1", "Column 3"], str)
sheet.apply_formatter(f)
```

can be replaced by:

```
sheet.column.format(["Column 1", "Column 3"], str)
```

2.11.6 Migrate from 0.2.1 to 0.2.2+

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 imported.
```

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.

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.

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*.

2.11.7 Migrate from 0.1.x to 0.2.x

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():
```

(continues on next page)

(continued from previous page)

```
...     print(line.rstrip())
1,2,3
4,5,6
```

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")
```

2.11.8 Change log

0.7.7 - 01.07.2026

Fixed

1. #310: fix broken indentation in iget_records memory-reduction patch

0.7.6 - 29.06.2026

Fixed

1. #307: honor negative indices in delete_rows and delete_columns

0.7.5 - 20.06.2026

Updated

1. migrate to pytest as nose is deprecated beyond python 3.9

Fixed

1. #296: provide thread-safe support for free_resources() and append
2. github copilot has found and correct lots of typos within the docs and docstrings. and it has contributed to the fix of 296.

0.7.4 - 25.10.2025

Fixed

1. #257: provide meaningful error messages where dest_* params are missing in save_as and isave_as

0.7.3 - 12.04.2025

Fixed

1. #263: support pathlib from python 3.4
2. #267: better error message for file_name

0.7.2 - 23.03.2025

Fixed

1. #270: apenddoc breaks interpreter optimization
2. #274: Replace deprecated imp module with importlib - enforced pyexcel's dependency on lml>=0.2.0. This will have an implication for linux distribution makers for pyexcel. However, this is a reaction to the potential removal of __import__ syntax.

0.7.1 - 11.09.2024

Fixed

1. #272: remove chardet as a requirement

Updated

1. various typos and linted the code

0.7.0 - 12.2.2022

Fixed

1. #250: RecursionError raised on deepcopy of a sheet

Updated

1. #255: pyexcel.get_array documentation page seems to be a copy of pyexcel.get_sheet

Removed

1. #249: drop the support for dummy import statements pyexcel.ext.*

0.6.7 - 12.09.2021

Updated

1. #243: fix small typo.
2. add chardet as explicit dependency

0.6.6 - 14.11.2020

Updated

1. #233: dynamically resize the table matrix on set_value. sheet['AA1'] = 'test' will work in this release.

0.6.5 - 8.10.2020

Updated

1. update queryset source to work with pyexcel-io 0.6.0

0.6.4 - 18.08.2020

Updated

1. #219: book created from dict no longer discards order.

0.6.3 - 01.08.2020

fixed

1. #214: remove leading and trailing whitespace for column names

removed

1. python 2 compatibility have been permanently removed.

0.6.2 - 8.06.2020

fixed

1. #109: Control the column order when write the data output

0.6.1 - 02.05.2020

fixed

1. #203: texttable was dropped out in 0.6.0 as compulsary dependency. end user may experience it when a sheet/table is printed in a shell. otherwise, new user of pyexcel won't see it. As of release date, no issues were created

0.6.0 - 21.04.2020

updated

1. #199: += in place; = + shall return new instance
2. #195: documentation update. however small is welcome

removed

1. Dropping the test support for python version lower than 3.6. v0.6.0 should work with python 2.7 but is not guaranteed to work. Please upgrade to python 3.6+.

0.5.15 - 07.07.2019

updated

1. #185: fix a bug with http data source. The real fix lies in pyexcel-io v0.5.19. this release just put the version requirement in.

0.5.14 - 12.06.2019

updated

1. #182: support dest_force_file_type on save_as and save_book_as

0.5.13 - 12.03.2019

updated

1. #176: get_sheet {IndexError}list index out of range // XLSX can't be opened

0.5.12 - 25.02.2019

updated

1. #174: include examples in tarbar

0.5.11 - 22.02.2019

updated

1. #169: remove pyexcel-handsontalbe in test
2. add tests, and docs folder in distribution

0.5.10 - 3.12.2018

updated

1. #157: Please use scan_plugins_regex, which lml 0.7 complains about
2. updated dependency on pyexcel-io to 0.5.11

0.5.9.1 - 30.08.2018

updated

1. to require pyexcel-io 0.5.9.1 and use lml at least version 0.0.2

0.5.9 - 30.08.2018

added

1. support __len__. len(book) returns the number of sheets and len(sheet) returns the number of rows
2. #144: memory-efficient way to read sheet names.
3. #148: force_file_type is introduced. When reading a file on a disk, this parameter allows you to choose a reader. i.e. csv reader for a text file. xlsx reader for a xlsx file but with .blob file suffix.
4. finally, pyexcel got import pyexcel.__version__

updated

1. Sheet.to_records() returns a generator now, saving memory
2. #115, Fix set membership test to run faster in python2
3. #140, Direct writes to cells yield weird results

0.5.8 - 26.03.2018

added

1. #125, sort book sheets

updated

1. #126, dest_sheet_name in save_as will set the sheet name in the output
2. #115, Fix set membership test to run faster in python2

0.5.7 - 11.01.2018

added

1. pyexcel-io#46, expose *bulk_save* to developer.

0.5.6 - 23.10.2017

removed

1. #105, remove gease from setup_requires, introduced by 0.5.5.
2. removed testing against python 2.6
3. #103, include LICENSE file in MANIFEST.in, meaning LICENSE file will appear in the released tar ball.

0.5.5 - 20.10.2017

removed

1. #105, remove gease from setup_requires, introduced by 0.5.5.
2. removed testing against python 2.6
3. #103, include LICENSE file in MANIFEST.in, meaning LICENSE file will appear in the released tar ball.

0.5.4 - 27.09.2017

fixed

1. #100, Sheet.to_dict() gets out of range error because there is only one row.

updated

1. Updated the baseline of pyexcel-io to 0.5.1.

0.5.3 - 01-08-2017

added

1. #95, respect the order of records in iget_records, isave_as and save_as.
2. #97, new feature to allow intuitive initialization of pyexcel.Book.

0.5.2 - 26-07-2017

Updated

1. embeded the enabler for pyexcel-htmlr. http source does not support text/html as mime type.

0.5.1 - 12.06.2017

Updated

1. support saving SheetStream and BookStream to database targets. This is needed for pyexcel-webio and its downstream projects.

0.5.0 - 19.06.2017

Added

1. Sheet.top() and Sheet.top_left() for data browsing
2. add html as default rich display in Jupyter notebook when pyexcel-text and pyexcel-chart is installed

3. add svg as default rich display in Jupyter notebook when pyexcel-chart and one of its implementation plugin(pyexcel-pygal, etc.) are is installed
4. new dictionary source supported: a dictionary of key value pair could be read into a sheet.
5. added dynamic external plugin loading. meaning if a pyexcel plugin is installed, it will be loaded implicitly. And this change would remove unnecessary info log for those who do not use pyexcel-text and pyexcel-gal
6. save_book_as before 0.5.0 becomes isave_book_as and save_book_as in 0.5.0 convert BookStream to Book before saving.
7. #83, file closing mechanism is enfored. free_resource is added and it should be called when iget_array, iget_records, isave_as and/or isave_book_as are used.

Updated

1. array is passed to pyexcel.Sheet as reference. it means your array data will be modified.

Removed

1. pyexcel.Writer and pyexcel.BookWriter were removed
2. pyexcel.load_book_from_sql and pyexcel.load_from_sql were removed
3. pyexcel.deprecated.load_from_query_sets, pyexcel.deprecated.load_book_from_django_models and pyexcel.deprecated.load_from_django_model were removed
4. Removed plugin loading code and lml is used instead

0.4.5 - 17.03.2017

Updated

1. #80: remove pyexcel-chart import from v0.4.x

0.4.4 - 06.02.2017

Updated

1. #68: regression save_to_memory() should have returned a stream instance which has been reset to zero if possible. The exception is sys.stdout, which cannot be reset.
2. #74: Not able to handle decimal.Decimal

Removed

1. remove get_{{file_type}}_stream functions from pyexcel.Sheet and pyexcel.Book introduced since 0.4.3.

0.4.3 - 26.01.2017

Added

1. '.stream' attribute are attached to *~pyexcel.Sheet* and *~pyexcel.Book* to get direct access the underneath stream in responding to file type attributes, such as sheet.xls. it helps provide a custom stream to external world, for example, Sheet.stream.csv gives a text stream that contains csv formatted data. Book.stream.xls returns a xls format data in a byte stream.

Updated

1. Better error reporting when an unknown parameters or unsupported file types were given to the signature functions.

0.4.2 - 17.01.2017

Updated

1. Raise exception if the incoming sheet does not have column names. In other words, only sheet with column names could be saved to database. sheet with row names cannot be saved. The alternative is to transpose the sheet, then name_columns_by_row and then save.
2. fix iget_records where a non-uniform content should be given, e.g. [["x", "y"], [1, 2], [3]], some record would become non-uniform, e.g. key 'y' would be missing from the second record.
3. *skip_empty_rows* is applicable when saving a python data structure to another data source. For example, if your array contains a row which is consisted of empty string, such as ['', '', '' ... ''], please specify *skip_empty_rows=False* in order to preserve it. This becomes subtle when you try save a python dictionary where empty rows is not easy to be spotted.
4. #69: better documentation for save_book_as.

0.4.1 - 23.12.2016

Updated

1. #68: regression save_to_memory() should have returned a stream instance.

0.4.0 - 22.12.2016

Added

1. Flask-Excel#19 allow sheet_name parameter
2. pyexcel-xls#11 case-insensitive for file_type. *xls* and *XLS* are treated in the same way

Updated

1. #66: *export_columns* is ignored
2. Update dependency on pyexcel-io v0.3.0

0.3.3 - 07.11.2016

Updated

1. #63: cannot display empty sheet(hence book with empty sheet) as texttable

0.3.2 - 02.11.2016

Updated

1. #62: optional module import error become visible.

0.3.0 - 28.10.2016

Added:

1. file type setters for Sheet and Book, and its documentation
2. *iget_records* returns a generator for a list of records and should have better memory performance, especially dealing with large csv files.
3. *iget_array* returns a generator for a list of two dimensional array and should have better memory performance, especially dealing with large csv files.
4. Enable pagination support, and custom row renderer via pyexcel-io v0.2.3

Updated

1. Take *isave_as* out from *save_as*. Hence two functions are there for save a sheet as
2. #60: encode 'utf-8' if the console is of ascii encoding.
3. #59: custom row renderer
4. #56: set cell value does not work
5. pyexcel.transpose becomes *pyexcel.sheets.transpose*
6. iterator functions of *pyexcel.Sheet* were converted to generator functions
   - *pyexcel.Sheet.enumerate()*
   - *pyexcel.Sheet.reverse()*
   - *pyexcel.Sheet.vertical()*
   - *pyexcel.Sheet.rvertical()*
   - *pyexcel.Sheet.rows()*
   - *pyexcel.Sheet.rrows()*
   - *pyexcel.Sheet.columns()*
   - *pyexcel.Sheet.rcolumns()*
   - *pyexcel.Sheet.named_rows()*
   - *pyexcel.Sheet.named_columns()*
7. *~pyexcel.Sheet.save_to_memory* and *~pyexcel.Book.save_to_memory* return the actual content. No longer they will return a io object hence you cannot call getvalue() on them.

Removed:

1. *content* and *out_file* as function parameters to the signature functions are no longer supported.
2. SourceFactory and RendererFactory are removed
3. The following methods are removed
   - *pyexcel.to_array*
   - *pyexcel.to_dict*
   - *pyexcel.utils.to_one_dimensional_array*
   - *pyexcel.dict_to_array*
   - *pyexcel.from_records*
   - *pyexcel.to_records*
4. *pyexcel.Sheet.filter* has been re-implemented and all filters were removed:
   - *pyexcel.filters.ColumnIndexFilter*
   - *pyexcel.filters.ColumnFilter*
   - *pyexcel.filters.RowFilter*
   - *pyexcel.filters.EvenColumnFilter*
   - *pyexcel.filters.OddColumnFilter*
   - *pyexcel.filters.EvenRowFilter*

- *pyexcel.filters.OddRowFilter*
- *pyexcel.filters.RowIndexFilter*
- *pyexcel.filters.SingleColumnFilter*
- *pyexcel.filters.RowValueFilter*
- *pyexcel.filters.NamedRowValueFilter*
- *pyexcel.filters.ColumnValueFilter*
- *pyexcel.filters.NamedColumnValueFilter*
- *pyexcel.filters.SingleRowFilter*

5. the following functions have been removed
   - *add_formatter*
   - *remove_formatter*
   - *clear_formatters*
   - *freeze_formatters*
   - *add_filter*
   - *remove_filter*
   - *clear_filters*
   - *freeze_formatters*
6. *pyexcel.Sheet.filter* has been re-implemented and all filters were removed:
   - pyexcel.formatters.SheetFormatter

0.2.5 - 31.08.2016

Updated:

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

0.2.4 - 14.07.2016

Updated:

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

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.

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")

0.2.1 - 23.04.2016

Added:

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

Updated:

1. code refactoring done for easy addition of sources.
2. bug fix #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#22

Removed:

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

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 improvements

0.1.7 - 03.07.2015

Added

1. Support pyramid-excel which does the database commit on its own.

0.1.6 - 13.06.2015

Added

1. get excel data from a http url

0.0.13 - 07.02.2015

Added

1. Support django
2. texttable as default renderer

0.0.12 - 25.01.2015

Added

1. Added sqlalchemy support

0.0.10 - 15.12.2015

Added

1. added csvz and tsvz format

0.0.4 - 12.10.2014

Updated

1. Support python 3

0.0.1 - 14.09.2014

Features:

1. read and write csv, ods, xls, xlsx and xlsm files(which are referred later as excel files)
2. various iterators for the reader
3. row and column filters for the reader
4. utilities to get array and dictionary out from excel files.
5. cookbook recipes for some common and simple usage of this library.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

__getitem__() (*pyexcel.Sheet method*), 157
__init__() (*pyexcel.Book method*), 143
__init__() (*pyexcel.Sheet method*), 153
__init__() (*pyexcel.internal.generators.BookStream method*), 174
__init__() (*pyexcel.internal.generators.SheetStream method*), 174
__init__() (*pyexcel.internal.sheets.Column method*), 176
__init__() (*pyexcel.internal.sheets.Matrix method*), 171
__init__() (*pyexcel.internal.sheets.Row method*), 175

A

array (*pyexcel.Sheet property*), 161

B

Book (*class in pyexcel*), 143
bookdict (*pyexcel.Book property*), 146
BookStream (*class in pyexcel.internal.generators*), 174

C

cell_value() (*pyexcel.Sheet method*), 157
colnames (*pyexcel.Sheet property*), 160
Column (*class in pyexcel.internal.sheets*), 176
column_at() (*pyexcel.Sheet method*), 158
column_range() (*pyexcel.Sheet method*), 156
content (*pyexcel.Sheet property*), 156
csv (*pyexcel.Book property*), 147
csv (*pyexcel.Sheet property*), 162
csvz (*pyexcel.Book property*), 147
csvz (*pyexcel.Sheet property*), 162
cut() (*pyexcel.Sheet method*), 168

D

delete_columns() (*pyexcel.Sheet method*), 158
delete_named_column_at() (*pyexcel.Sheet method*), 159
delete_named_row_at() (*pyexcel.Sheet method*), 160
delete_rows() (*pyexcel.Sheet method*), 157
dict (*pyexcel.Sheet property*), 161

E

extend_columns() (*pyexcel.Sheet method*), 158
extend_rows() (*pyexcel.Sheet method*), 157
extract_a_sheet_from_a_book() (*in module pyexcel*), 142

F

filter() (*pyexcel.Sheet method*), 166
format() (*pyexcel.Sheet method*), 165
free_resources() (*in module pyexcel*), 124

G

get_array() (*in module pyexcel*), 92
get_book() (*in module pyexcel*), 107
get_book_dict() (*in module pyexcel*), 105
get_dict() (*in module pyexcel*), 95
get_records() (*in module pyexcel*), 100
get_sheet() (*in module pyexcel*), 109

I

iget_array() (*in module pyexcel*), 116
iget_book() (*in module pyexcel*), 114
iget_records() (*in module pyexcel*), 119
isave_as() (*in module pyexcel*), 130
isave_book_as() (*in module pyexcel*), 139

M

map() (*pyexcel.Sheet method*), 167
Matrix (*class in pyexcel.internal.sheets*), 171
merge_all_to_a_book() (*in module pyexcel*), 142
merge_csv_to_a_book() (*in module pyexcel*), 142

N

name_columns_by_row() (*pyexcel.Sheet method*), 159
name_rows_by_column() (*pyexcel.Sheet method*), 160
named_column_at() (*pyexcel.Sheet method*), 159
named_row_at() (*pyexcel.Sheet method*), 160
number_of_columns() (*pyexcel.Sheet method*), 156
number_of_rows() (*pyexcel.Sheet method*), 156

number_of_sheets() (*pyexcel.Book method*), 146

O

ods (*pyexcel.Book property*), 149
ods (*pyexcel.Sheet property*), 164

P

paste() (*pyexcel.Sheet method*), 168
project() (*pyexcel.Sheet method*), 166

R

records (*pyexcel.Sheet property*), 161
region() (*pyexcel.Sheet method*), 168
Row (*class in pyexcel.internal.sheets*), 175
row_at() (*pyexcel.Sheet method*), 157
row_range() (*pyexcel.Sheet method*), 156
rownames (*pyexcel.Sheet property*), 159

S

save_as() (*in module pyexcel*), 124
save_as() (*pyexcel.Book method*), 150
save_as() (*pyexcel.Sheet method*), 169
save_book_as() (*in module pyexcel*), 136
save_to_database() (*pyexcel.Book method*), 151
save_to_database() (*pyexcel.Sheet method*), 170
save_to_django_model() (*pyexcel.Sheet method*), 171
save_to_django_models() (*pyexcel.Book method*), 151
save_to_memory() (*pyexcel.Book method*), 151
save_to_memory() (*pyexcel.Sheet method*), 170
set_column_at() (*pyexcel.Sheet method*), 158
set_named_column_at() (*pyexcel.Sheet method*), 159
set_named_row_at() (*pyexcel.Sheet method*), 160
set_row_at() (*pyexcel.Sheet method*), 157
Sheet (*class in pyexcel*), 152
sheet_names() (*pyexcel.Book method*), 146
SheetStream (*class in pyexcel.internal.generators*), 174
split_a_book() (*in module pyexcel*), 142
stream (*pyexcel.Book property*), 149
stream (*pyexcel.Sheet property*), 164

T

transpose() (*pyexcel.Sheet method*), 167
tsv (*pyexcel.Book property*), 147
tsv (*pyexcel.Sheet property*), 162
tsvz (*pyexcel.Book property*), 148
tsvz (*pyexcel.Sheet property*), 163

U

url (*pyexcel.Book property*), 146
url (*pyexcel.Sheet property*), 162

X

xls (*pyexcel.Book property*), 148
xls (*pyexcel.Sheet property*), 163
xlsm (*pyexcel.Book property*), 148
xlsm (*pyexcel.Sheet property*), 163
xlsx (*pyexcel.Book property*), 149
xlsx (*pyexcel.Sheet property*), 164