pyexcel-io

Release 0.6.6

Jan 29, 2022

Migration Note

1 Introduction **3**

2 Installation **5**

3 Plugin shopping guide **7**
- 3.1 Packaging with PyInstaller . . . 8
- 3.2 Working with CSV format . . . 9
- 3.3 Read partial data . . . 12
- 3.4 Rendering(Formatting) the data . . . 13
- 3.5 Saving multiple sheets as CSV format . . . 13
- 3.6 File formats: .csvz and .tsvz . . . 14
- 3.7 Working with sqlalchemy . . . 16
- 3.8 Working with django database . . . 20
- 3.9 Extend pyexcel-io for other excel or tabular formats . . . 22

4 API **27**
- 4.1 Common parameters . . . 27

5 Indices and tables **29**

Author C.W.

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

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

License New BSD License

Released 0.6.6

Generated Jan 29, 2022

CHAPTER 1

Introduction

pyexcel-io provides **one** application programming interface(API) to read and write data in different excel formats. It makes information processing involving excel files a simple task. The data in excel files can be turned into an ordered dictionary with least code. This library focuses on data processing using excel files as storage media hence fonts, colors and charts were not and will not be considered.

It was created due to the lack of uniform programming interface to access data in different excel formats. A developer needs to use different methods of different libraries to read the same data in different excel formats, hence the resulting code is cluttered and unmaintainable. This is a challenge posed by users who do not know or care about the differences in excel file formats. Instead of educating the users about the specific excel format a data processing application supports, the library takes up the challenge and promises to support all known excel formats.

All great work have done by individual library developers. This library unites only the data access API. With that said, **pyexcel-io** also bring something new on the table: *"csvz" and "tsvz"* format, new format names as of 2014. They are invented and supported by pyexcel-io.

CHAPTER 2

Installation

You can install pyexcel-io via pip:

```
$ pip install pyexcel-io
```

or clone it and install it:

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

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]	
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

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

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

CHAPTER 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').

After that, you can start get and save data in the loaded format. There are two plugins for the same file format, e.g. pyexcel-ods3 and pyexcel-ods. If you want to choose one, please try pip uninstall the un-wanted one. And if you want to have both installed but wanted to use one of them for a function call(or file type) and the other for another function call(or file type), you can pass on "library" option to get_data and save_data, e.g. get_data(.., library='pyexcel-ods')

Note: pyexcel-text is no longer a plugin of pyexcel-io but a direct plugin of pyexcel

Table 1: Plugin compatibility table

pyexcel-io	xls	xlsx	ods	ods3	odsr	xlsxw
0.6.0+	0.5.0+	0.5.0+	0.5.4	0.5.3	0.5.0+	0.5.0+
0.5.10+	0.5.0+	0.5.0+	0.5.4	0.5.3	0.5.0+	0.5.0+
0.5.1+	0.5.0+	0.5.0+	0.5.0+	0.5.0+	0.5.0+	0.5.0+
0.4.x	0.4.x	0.4.x	0.4.x	0.4.x	0.4.x	0.4.x
0.3.0+	0.3.0+	0.3.0	0.3.0+	0.3.0+	0.3.0	0.3.0
0.2.2+	0.2.2+	0.2.2+	0.2.1+	0.2.1+		0.0.1
0.2.0+	0.2.0+	0.2.0+	0.2.0	0.2.0		0.0.1

3.1 Packaging with PyInstaller

With pyexcel-io v0.6.0, the way to package it has been changed because plugin interface update.

3.1.1 Built-in plugins for pyexcel-io

In order to package every built-in plugins of pyexcel-io, you need to specify:

```
--hidden-import pyexcel_io.readers.csv_in_file
--hidden-import pyexcel_io.readers.csv_in_memory
--hidden-import pyexcel_io.readers.csv_content
--hidden-import pyexcel_io.readers.csvz
--hidden-import pyexcel_io.writers.csv_in_file
--hidden-import pyexcel_io.writers.csv_in_memory
--hidden-import pyexcel_io.writers.csvz_writer
--hidden-import pyexcel_io.database.importers.django
--hidden-import pyexcel_io.database.importers.sqlalchemy
--hidden-import pyexcel_io.database.exporters.django
--hidden-import pyexcel_io.database.exporters.sqlalchemy
```

With pyexcel-io v0.4.0, the way to package it has been changed because it uses lml for all plugins.

3.1.2 Built-in plugins of pyexcel-io

In order to package every built-in plugins of pyexcel-io, you need to specify:

```
--hidden-import pyexcel_io.readers.csvr
--hidden-import pyexcel_io.readers.csvz
--hidden-import pyexcel_io.readers.tsv
--hidden-import pyexcel_io.readers.tsvz
--hidden-import pyexcel_io.writers.csvw
--hidden-import pyexcel_io.writers.csvz
--hidden-import pyexcel_io.writers.tsv
--hidden-import pyexcel_io.writers.tsvz
--hidden-import pyexcel_io.database.importers.django
--hidden-import pyexcel_io.database.importers.sqlalchemy
--hidden-import pyexcel_io.database.exporters.django
--hidden-import pyexcel_io.database.exporters.sqlalchemy
```

3.1.3 pyexcel-xlsx

In order to package pyexcel-xlsx, you need to specify:

```
--hidden-import pyexcel_xlsx
--hidden-import pyexcel_xlsx.xlsxr
--hidden-import pyexcel_xlsx.xlsxw
```

3.1.4 pyexcel-xlsxw

In order to package pyexcel-xlsxw, you need to specify:

```
--hidden-import pyexcel_xlsxw
--hidden-import pyexcel_xlsxw.xlsxw
```

3.1.5 pyexcel-xls

In order to package pyexcel-xls, you need to specify:

```
--hidden-import pyexcel_xls
--hidden-import pyexcel_xls.xlsr
--hidden-import pyexcel_xls.xlsw
```

3.1.6 pyexcel-ods

In order to package pyexcel-ods, you need to specify:

```
--hidden-import pyexcel_ods
--hidden-import pyexcel_ods.odsr
--hidden-import pyexcel_ods.odsw
```

3.1.7 pyexcel-ods3

In order to package pyexcel-ods3, you need to specify:

```
--hidden-import pyexcel_ods3
--hidden-import pyexcel_ods3.odsr
--hidden-import pyexcel_ods3.odsw
```

3.1.8 pyexcel-odsr

In order to package pyexcel-odsr, you need to specify:

```
--hidden-import pyexcel_odsr
--hidden-import pyexcel_odsr.odsr
```

3.2 Working with CSV format

Please note that csv reader load data in a lazy manner. It ignores excessive trailing cells that has None value. For example, the following csv content:

```
1,2,,,,,
3,4,,,,,
5,,,,,,,
```

would end up as:

```
1,2
3,4
5,
```

3.2.1 Write to a csv file

Here's the sample code to write an array to a csv file

```
>>> import datetime
>>> from pyexcel_io import save_data
>>> data = [
...     [1, 2.0, 3.0],
...     [
...         datetime.date(2016, 5, 4),
...         datetime.datetime(2016, 5, 4, 17, 39, 12),
...         datetime.datetime(2016, 5, 4, 17, 40, 12, 100)
...     ]
... ]
>>> save_data("your_file.csv", data)
```

Let's verify the file content:

```
>>> with open("your_file.csv", "r") as csvfile:
...     for line in csvfile.readlines():
...         print(line.strip())
1,2.0,3.0
2016-05-04,2016-05-04 17:39:12,2016-05-04 17:40:12.000100
```

Change line endings

By default, python csv module provides windows line ending 'rn'. In order to change it, you can do:

```
>>> save_data("your_file.csv", data, lineterminator='\n')
```

3.2.2 Read from a csv file

And we can read the written csv file back as the following code:

```
>>> from pyexcel_io import get_data
>>> import pprint
>>> data = get_data("your_file.csv")
>>> pprint.pprint(data['your_file.csv'])
[[1, 2.0, 3.0],
 [datetime.date(2016, 5, 4),
  datetime.datetime(2016, 5, 4, 17, 39, 12),
  datetime.datetime(2016, 5, 4, 17, 40, 12, 100)]]
```

As you can see, pyexcel-io not only reads the csv file back but also recognizes the data types: *int*, *float*, *date* and *datetime*. However, it does give your cpu some extra job. When you are handling a large csv file and the cpu budget is of your concern, you may switch off the type detection feature. For example, let's switch all off:

```
>>> data = get_data("your_file.csv", auto_detect_float=False, auto_detect_
↪datetime=False)
    >>> import json
>>> json.dumps(data['your_file.csv'])
'[[1, "2.0", "3.0"], ["2016-05-04", "2016-05-04 17:39:12", "2016-05-04 17:40:12.000100
↪"]]'
```

In addition to *auto_detect_float* and *auto_detect_datetime*, there is another flag named *auto_detect_int*, which becomes active only if *auto_detect_float* is *True*. Now, let's play a bit with *auto_detect_int*:

```
>>> data = get_data("your_file.csv", auto_detect_int=False)
>>> pprint.pprint(data['your_file.csv'])
[[1.0, 2.0, 3.0],
 [datetime.date(2016, 5, 4),
  datetime.datetime(2016, 5, 4, 17, 39, 12),
  datetime.datetime(2016, 5, 4, 17, 40, 12, 100)]]
```

As you see, all numeric data are identified as float type. If you looked a few paragraphs above, you would notice *auto_detect_int* affected [1, 2, ..] in the first row.

3.2.3 Write a csv to memory

Here's the sample code to write a dictionary as a csv into memory:

```
>>> from pyexcel_io import save_data
>>> data = [[1, 2, 3], [4, 5, 6]]
>>> io = StringIO()
>>> save_data(io, data)
>>> # do something with the io
>>> # In reality, you might give it to your http response
>>> # object for downloading
```

3.2.4 Read from a csv from memory

Continue from previous example:

```
>>> # This is just an illustration
>>> # In reality, you might deal with csv file upload
>>> # where you will read from requests.FILES['YOUR_XL_FILE']
>>> import json
>>> data = get_data(io)
>>> print(json.dumps(data))
{"csv": [[1, 2, 3], [4, 5, 6]]}
```

3.2.5 Encoding parameter

In general, if you would like to save your csv file into a custom encoding, you can specify 'encoding' parameter. Here is how you write verses of a finnish song, "Aurinko laskee länteen"[1] into a csv file

```
>>> content = [[u'Aurinko laskee länteen', u'Näin sen ja ymmärsin sen', u'Poissa aika␣
↪on rakkauden Kun aurinko laskee länteen']]
>>> test_file = "test-utf16-encoding.csv"
    >>> save_data(test_file, content, encoding="utf-16", lineterminator="\n")
```

In the reverse direction, if you would like to read your csv file with custom encoding back, you do the same to get_data:

```
>>> custom_encoded_content = get_data(test_file, encoding="utf-16")
    >>> assert custom_encoded_content[test_file] == content
```

[1] A finnish song that was entered in Eurovision in 1965. You can check out its lyrics at diggiloo.net

3.2.6 Byte order mark (BOM) in csv file

By passing **encoding="utf-8-sig", You can write UTF-8 BOM header into your csv file. Here is an example to write a sentence of "Shui Dial Getou"[#f2] into a csv file:

```
>>> content = [['löyly', 'löyly']]
>>> test_file = "test-utf8-BOM.csv"
    >>> save_data(test_file, content, encoding="utf-8-sig", lineterminator="\n")
```

When you read it back you will have to specify encoding too.

```
>>> custom_encoded_content = get_data(test_file, encoding="utf-8-sig")
    >>> assert custom_encoded_content[test_file] == content
```

3.3 Read partial data

When you are dealing with huge amount of data, obviously you would not like to fill up your memory with those data. Here is a the feature to support pagination of your data.

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

```
>>> import datetime
>>> from pyexcel_io import save_data
>>> data = [
...     [1, 21, 31],
...     [2, 22, 32],
...     [3, 23, 33],
...     [4, 24, 34],
...     [5, 25, 35],
...     [6, 26, 36]
... ]
>>> save_data("your_file.csv", data)
```

And let's pretend to read partial data:

```
>>> from pyexcel_io import get_data
>>> data = get_data("your_file.csv", start_row=2, row_limit=3)
>>> data['your_file.csv']
[[3, 23, 33], [4, 24, 34], [5, 25, 35]]
```

And you could as well do the same for columns:

```
>>> data = get_data("your_file.csv", start_column=1, column_limit=2)
>>> data['your_file.csv']
[[21, 31], [22, 32], [23, 33], [24, 34], [25, 35], [26, 36]]
```

Obvious, you could do both at the same time:

```
>>> data = get_data("your_file.csv",
...     start_row=2, row_limit=3,
...     start_column=1, column_limit=2)
>>> data['your_file.csv']
[[23, 33], [24, 34], [25, 35]]
```

The pagination support is available across all pyexcel-io plugins.

3.4 Rendering(Formatting) the data

You might want to do custom rendering on your data obtained. *row_renderer* was added since version 0.2.3. Here is how you can use it.

Let's assume the following file:

```
>>> import datetime
>>> from pyexcel_io import save_data
>>> data = [
...     [1, 21, 31],
...     [2, 22, 32],
...     [3, 23, 33]
... ]
>>> save_data("your_file.csv", data)
```

And let's read them back:

```
>>> from pyexcel_io import get_data
>>> data = get_data("your_file.csv")
>>> data['your_file.csv']
[[1, 21, 31], [2, 22, 32], [3, 23, 33]]
```

And you may want use row_renderer to customize it to string:

```
>>> def my_renderer(row):
...     return [str(element) for element in row]
>>> data = get_data("your_file.csv", row_renderer=my_renderer)
>>> data['your_file.csv']
[['1', '21', '31'], ['2', '22', '32'], ['3', '23', '33']]
```

3.5 Saving multiple sheets as CSV format

3.5.1 Write to multiple sibling csv files

Here's the sample code to write a dictionary to multiple sibling csv files:

```
>>> from pyexcel_io import save_data
>>> data = OrderedDict() # from collections import OrderedDict
>>> data.update({"Sheet 1": [[1, 2, 3], [4, 5, 6]]})
>>> data.update({"Sheet 2": [["row 1", "row 2", "row 3"]]})
>>> save_data("your_file.csv", data)
```

3.5.2 Read from multiple sibling csv files

Here's the sample code:

```
>>> from pyexcel_io import get_data
>>> data = get_data("your_file.csv")
>>> import json
>>> print(json.dumps(data))
{"Sheet 1": [[1, 2, 3], [4, 5, 6]], "Sheet 2": [["row 1", "row 2", "row 3"]]}
```

Here is what you would get:

```
>>> import glob
>>> list = glob.glob("your_file__*.csv")
>>> json.dumps(sorted(list))
'["your_file__Sheet 1__0.csv", "your_file__Sheet 2__1.csv"]'
```

3.5.3 Write multiple sibling csv files to memory

Here's the sample code to write a dictionary of named two dimensional array into memory:

```
>>> from pyexcel_io import save_data
>>> data = OrderedDict()
>>> data.update({"Sheet 1": [[1, 2, 3], [4, 5, 6]]})
>>> data.update({"Sheet 2": [[7, 8, 9], [10, 11, 12]]})
>>> io = StringIO()
>>> save_data(io, data)
>>> # do something with the io
>>> # In reality, you might give it to your http response
>>> # object for downloading
```

3.5.4 Read multiple sibling csv files from memory

Continue from previous example:

```
>>> # This is just an illustration
>>> # In reality, you might deal with csv file upload
>>> # where you will read from requests.FILES['YOUR_XL_FILE']
>>> data = get_data(io, multiple_sheets=True)
>>> print(json.dumps(data))
{"Sheet 1": [[1, 2, 3], [4, 5, 6]], "Sheet 2": [[7, 8, 9], [10, 11, 12]]}
```

3.6 File formats: .csvz and .tsvz

3.6.1 Introduction

'csvz' and 'tsvz' are newly invented excel file formats by pyexcel. Simply put, 'csvz' is the zipped content of one or more csv file(s). 'tsvz' is the twin brother of 'csvz'. They are similiar to the implementation of xlsx format, which is a zip of excel content in xml format.

The obvious tangile benefit of zipped csv over normal csv is the reduced file size. However, the drawback is the need of unzipping software.

3.6.2 Single Sheet

When a single sheet is to be saved, the resulting csvz file will be a zip file that contains one csv file bearing the name of `Sheet`.

```
>>> from pyexcel_io import save_data
>>> data = [[1,2,3]]
>>> save_data("myfile.csvz", data)
>>> import zipfile
>>> zip = zipfile.ZipFile("myfile.csvz", 'r')
>>> zip.namelist()
['pyexcel_sheet1.csv']
>>> zip.close()
```

And it can be read out as well and can be saved in any other supported format.

```
>>> from pyexcel_io import get_data
>>> data = get_data("myfile.csvz")
>>> import json
>>> json.dumps(data)
'{"pyexcel_sheet1": [[1, 2, 3]]}'
```

3.6.3 Multiple Sheet Book

When multiple sheets are to be saved as a book, the resulting csvz file will be a zip file that contains each sheet as a csv file name

```
>>> from pyexcel_io._compact import OrderedDict
>>> content = OrderedDict()
>>> content.update({
...     'Sheet 1':
...         [
...             [1.0, 2.0, 3.0],
...             [4.0, 5.0, 6.0],
...             [7.0, 8.0, 9.0]
...         ]
... })
>>> content.update({
...     'Sheet 2':
...         [
...             ['X', 'Y', 'Z'],
...             [1.0, 2.0, 3.0],
...             [4.0, 5.0, 6.0]
...         ]
... })
>>> content.update({
...     'Sheet 3':
...         [
...             ['O', 'P', 'Q'],
...             [3.0, 2.0, 1.0],
...             [4.0, 3.0, 2.0]
...         ]
... })
>>> save_data("mybook.csvz", content)
>>> import zipfile
>>> zip = zipfile.ZipFile("mybook.csvz", 'r')
>>> zip.namelist()
['Sheet 1.csv', 'Sheet 2.csv', 'Sheet 3.csv']
>>> zip.close()
```

The csvz book can be read back with two lines of code. And once it is read out, it can be saved in any other supported

format.

```
>>> book2 = get_data("mybook.csvz")
>>> json.dumps(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]]}'
```

3.6.4 Open csvz without pyexcel-io

All you need is a unzipping software. I would recommend 7zip which is open source and is available on all available OS platforms.

On latest Windows platform (windows 8), zip file is supported so just give the "csvz" file a file extension as ".zip". The file can be opened by File Explorer.

3.7 Working with sqlalchemy

Suppose we have a pure sql database connection via sqlalchemy:

```
>>> 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:///sqlalchemy.db")
>>> Base=declarative_base()
>>> Session=sessionmaker(bind=engine)
```

3.7.1 Write data to a database table

Assume we have the following database table:

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

Let's clear the database and create previous table in the database:

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

And suppose we have the following data structure to be saved:

```
>>> import datetime
>>> data = [
...     ['birth', 'id', 'name', 'weight'],
...     [datetime.date(2014, 11, 11), 0, 'Adam', 11.25],
...     [datetime.date(2014, 11, 12), 1, 'Smith', 12.25]
... ]
```

Here's the actual code to achieve it:

```
>>> from pyexcel_io import save_data
>>> from pyexcel_io.constants import DB_SQL, DEFAULT_SHEET_NAME
>>> from pyexcel_io.database.common import SQLTableImporter, SQLTableImportAdapter
>>> mysession = Session()
>>> importer = SQLTableImporter(mysession)
>>> adapter = SQLTableImportAdapter(Pyexcel)
>>> adapter.column_names = data[0]
>>> importer.append(adapter)
>>> save_data(importer, {adapter.get_name(): data[1:]}, file_type=DB_SQL)
```

Please note that, the data dict shall have table name as its key. Now let's verify the data:

```
>>> from pyexcel_io.database.querysets import QuerysetsReader
>>> query_sets=mysession.query(Pyexcel).all()
    >>> reader = QuerysetsReader(query_sets, data[0])
>>> results = reader.to_array()
>>> import json
>>> json.dumps(list(results))
'[["birth", "id", "name", "weight"], ["2014-11-11", 0, "Adam", 11.25], ["2014-11-12",␣
↪1, "Smith", 12.25]]'
```

3.7.2 Read data from a database table

Let's use previous data for reading and see if we could get them via `get_data()` :

```
>>> from pyexcel_io import get_data
>>> from pyexcel_io.database.common import SQLTableExporter, SQLTableExportAdapter
>>> exporter = SQLTableExporter(mysession)
>>> adapter = SQLTableExportAdapter(Pyexcel)
>>> exporter.append(adapter)
>>> data = get_data(exporter, file_type=DB_SQL)
>>> json.dumps(list(data['pyexcel']))
'[["birth", "id", "name", "weight"], ["2014-11-11", 0, "Adam", 11.25], ["2014-11-12",␣
↪1, "Smith", 12.25]]'
```

Read a subset from the table:

```
>>> exporter = SQLTableExporter(mysession)
>>> adapter = SQLTableExportAdapter(Pyexcel, ['birth'])
>>> exporter.append(adapter)
>>> data = get_data(exporter, file_type=DB_SQL)
>>> json.dumps(list(data['pyexcel']))
'[["birth"], ["2014-11-11"], ["2014-11-12"]]'
```

3.7.3 Write data to multiple tables

Before we start, let's clear off previous table:

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

Now suppose we have these more complex tables:

```
>>> from sqlalchemy import ForeignKey, DateTime
>>> from sqlalchemy.orm import relationship, backref
```

(continues on next page)

(continued from previous page)

```
>>> import sys
>>> class Post(Base):
...     __tablename__ = 'post'
...     id = Column(Integer, primary_key=True)
...     title = Column(String(80))
...     body = Column(String(100))
...     pub_date = Column(DateTime)
...
...     category_id = Column(Integer, ForeignKey('category.id'))
...     category = relationship('Category',
...         backref=backref('posts', lazy='dynamic'))
...
...     def __init__(self, title, body, category, pub_date=None):
...         self.title = title
...         self.body = body
...         if pub_date is None:
...             pub_date = datetime.utcnow()
...         self.pub_date = pub_date
...         self.category = category
...
...     def __repr__(self):
...         return '<Post %r>' % self.title
...
>>> class Category(Base):
...     __tablename__ = 'category'
...     id = Column(Integer, primary_key=True)
...     name = Column(String(50))
...
...     def __init__(self, name):
...         self.name = name
...
...     def __repr__(self):
...         return '<Category %r>' % self.name
...     def __str__(self):
...         return self.__repr__()
```

Let's clear the database and create previous table in the database:

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

Suppose we have these data:

```
>>> data = {
...     "Category":[
...         ["id", "name"],
...         [1, "News"],
...         [2, "Sports"]
...     ],
...     "Post":[
...         ["id", "title", "body", "pub_date", "category"],
...         [1, "Title A", "formal", datetime.datetime(2015,1,20,23,28,29), "News"],
...         [2, "Title B", "informal", datetime.datetime(2015,1,20,23,28,30), "Sports
↪"]
...     ]
...  }
```

Both table has gotten initialization functions:

```
>>> def category_init_func(row):
...     c = Category(row['name'])
...     c.id = row['id']
...     return c
```

and particularly **Post** has a foreign key to **Category**, so we need to query **Category** out and assign it to **Post** instance

```
>>> def post_init_func(row):
...     c = mysession.query(Category).filter_by(name=row['category']).first()
...     p = Post(row['title'], row['body'], c, row['pub_date'])
...     return p
```

Here's the code to update both:

```
>>> tables = {
...     "Category": [Category, data['Category'][0], None, category_init_func],
...     "Post": [Post, data['Post'][0], None, post_init_func]
... }
>>> from pyexcel_io._compact import OrderedDict
>>> importer = SQLTableImporter(mysession)
>>> adapter1 = SQLTableImportAdapter(Category)
>>> adapter1.column_names = data['Category'][0]
>>> adapter1.row_initializer = category_init_func
>>> importer.append(adapter1)
>>> adapter2 = SQLTableImportAdapter(Post)
>>> adapter2.column_names = data['Post'][0]
>>> adapter2.row_initializer = post_init_func
>>> importer.append(adapter2)
>>> to_store = OrderedDict()
>>> to_store.update({adapter1.get_name(): data['Category'][1:]})
>>> to_store.update({adapter2.get_name(): data['Post'][1:]})
>>> save_data(importer, to_store, file_type=DB_SQL)
```

Let's verify what do we have in the database:

```
>>> query_sets = mysession.query(Category).all()
>>> reader = QuerysetsReader(query_sets, data['Category'][0])
    >>> results = reader.to_array()
>>> import json
>>> json.dumps(list(results))
'[["id", "name"], [1, "News"], [2, "Sports"]]'
>>> query_sets = mysession.query(Post).all()
>>> reader = QuerysetsReader(query_sets, ["id", "title", "body", "pub_date"])
    >>> results = reader.to_array()
>>> json.dumps(list(results))
'[["id", "title", "body", "pub_date"], [1, "Title A", "formal", "2015-01-20T23:28:29
↪"], [2, "Title B", "informal", "2015-01-20T23:28:30"]]'
```

Skipping existing record

When you import data into a database that has data already, you can skip existing record if `pyexcel_io.PyexcelSQLSkipRowException` is raised. Example can be found here in test code.

Update existing record

When you import data into a database that has data already, you can update an existing record if you can query it from the database and set the data yourself and most importantly return it. You can find an example in test skipping row

3.7.4 Read data from multiple tables

Let's use previous data for reading and see if we could get them via `get_data()` :

```
>>> exporter = SQLTableExporter(mysession)
>>> adapter = SQLTableExportAdapter(Category)
>>> exporter.append(adapter)
>>> adapter = SQLTableExportAdapter(Post)
>>> exporter.append(adapter)
>>> data = get_data(exporter, file_type=DB_SQL)
>>> json.dumps(data)
'{"category": [["id", "name"], [1, "News"], [2, "Sports"]], "post": [["body",
↪"category_id", "id", "pub_date", "title"], ["formal", 1, 1, "2015-01-20T23:28:29",
↪"Title A"], ["informal", 2, 2, "2015-01-20T23:28:30", "Title B"]]}'
```

What if we read a subset per each table

```
>>> exporter = SQLTableExporter(mysession)
>>> adapter = SQLTableExportAdapter(Category, ['name'])
>>> exporter.append(adapter)
>>> adapter = SQLTableExportAdapter(Post, ['title'])
>>> exporter.append(adapter)
>>> data = get_data(exporter, file_type=DB_SQL)
>>> json.dumps(data)
'{"category": [["name"], ["News"], ["Sports"]], "post": [["title"], ["Title A"], [
↪"Title B"]]}'
```

3.8 Working with django database

This section shows the way to to write and read from django database. Becuase it is "heavy"" to include a django site here to show you. A mocked django model is used here to demonstate it:

```
>>> class FakeDjangoModel:
...     def __init__(self):
...         self.objects = Objects()
...         self._meta = Meta()
...
...     def __call__(self, **keywords):
...         return keywords
```

Note: You can visit django-excel documentation if you would prefer a real django model to be used in tutorial.

3.8.1 Write data to a django model

Let's suppose we have a django model:

```
>>> from pyexcel_io import save_data
>>> from pyexcel_io.constants import DB_DJANGO, DEFAULT_SHEET_NAME
>>> from pyexcel_io.database.common import DjangoModelImporter,␣
↪DjangoModelImportAdapter
>>> from pyexcel_io.database.common import DjangoModelExporter,␣
↪DjangoModelExportAdapter
>>> model = FakeDjangoModel()
```

Suppose you have these data:

```
>>> data  = [
...     ["X", "Y", "Z"],
...     [1, 2, 3],
...     [4, 5, 6]
... ]
>>> importer = DjangoModelImporter()
>>> adapter = DjangoModelImportAdapter(model)
>>> adapter.column_names = data[0]
>>> importer.append(adapter)
>>> save_data(importer, {adapter.get_name(): data[1:]}, file_type=DB_DJANGO)
>>> import pprint
>>> pprint.pprint(model.objects.objs)
[{'X': 1, 'Y': 2, 'Z': 3}, {'X': 4, 'Y': 5, 'Z': 6}]
```

3.8.2 Read data from a django model

Continue from previous example, you can read this back:

```
>>> from pyexcel_io import get_data
>>> exporter = DjangoModelExporter()
>>> adapter = DjangoModelExportAdapter(model)
>>> exporter.append(adapter)
>>> data = get_data(exporter, file_type=DB_DJANGO)
>>> data
OrderedDict([('Sheet0', [['X', 'Y', 'Z'], [1, 2, 3], [4, 5, 6]])])
```

Read a sub set of the columns:

```
>>> exporter = DjangoModelExporter()
>>> adapter = DjangoModelExportAdapter(model, ['X'])
>>> exporter.append(adapter)
>>> data = get_data(exporter, file_type=DB_DJANGO)
>>> data
OrderedDict([('Sheet0', [['X'], [1], [4]])])
```

3.8.3 Write data into multiple models

Suppose you have the following data to be stored in the database:

```
>>> data = {
...     "Sheet1": [['X', 'Y', 'Z'], [1, 4, 7], [2, 5, 8], [3, 6, 9]],
...     "Sheet2": [['A', 'B', 'C'], [1, 4, 7], [2, 5, 8], [3, 6, 9]]
... }
```

And want to save them to two django models:

```
>>> model1 = FakeDjangoModel()
>>> model2 = FakeDjangoModel()
```

In order to store a dictionary data structure, you need to do some transformation:

```
>>> importer = DjangoModelImporter()
>>> adapter1 = DjangoModelImportAdapter(model1)
>>> adapter1.column_names = data['Sheet1'][0]
>>> adapter2 = DjangoModelImportAdapter(model2)
>>> adapter2.column_names = data['Sheet2'][0]
>>> importer.append(adapter1)
>>> importer.append(adapter2)
>>> to_store = {
...     adapter1.get_name(): data['Sheet1'][1:],
...     adapter2.get_name(): data['Sheet2'][1:]
... }
>>> save_data(importer, to_store, file_type=DB_DJANGO)
>>> pprint.pprint(model1.objects.objs)
[{'X': 1, 'Y': 4, 'Z': 7}, {'X': 2, 'Y': 5, 'Z': 8}, {'X': 3, 'Y': 6, 'Z': 9}]
>>> pprint.pprint(model2.objects.objs)
[{'A': 1, 'B': 4, 'C': 7}, {'A': 2, 'B': 5, 'C': 8}, {'A': 3, 'B': 6, 'C': 9}]
```

3.8.4 Read content from multiple tables

Here's what you need to do:

```
>>> exporter = DjangoModelExporter()
>>> adapter1 = DjangoModelExportAdapter(model1)
>>> adapter2 = DjangoModelExportAdapter(model2)
>>> exporter.append(adapter1)
>>> exporter.append(adapter2)
>>> data = get_data(exporter, file_type=DB_DJANGO)
>>> data
OrderedDict([('Sheet1', [['X', 'Y', 'Z'], [1, 4, 7], [2, 5, 8], [3, 6, 9]]), ('Sheet2
↪', [['A', 'B', 'C'], [1, 4, 7], [2, 5, 8], [3, 6, 9]])])
```

What if we need only a subset of each model

```
>>> exporter = DjangoModelExporter()
>>> adapter1 = DjangoModelExportAdapter(model1, ['X'])
>>> adapter2 = DjangoModelExportAdapter(model2, ['A'])
>>> exporter.append(adapter1)
>>> exporter.append(adapter2)
>>> data = get_data(exporter, file_type=DB_DJANGO)
>>> data
OrderedDict([('Sheet1', [['X'], [1], [2], [3]]), ('Sheet2', [['A'], [1], [2], [3]])])
```

3.9 Extend pyexcel-io for other excel or tabular formats

You are welcome to extend pyexcel-io to read and write more tabular formats. No. 1 rule, your plugin must have a prefix '**pyexcel_**' in its module path. For example, *pyexcel-xls* has 'pyexcel_xls' as its module path. Otherwise, pyexcel-io will not load your plugin.

On github, you will find two examples in *examples* folder. This section explains its implementations to help you write yours.

Note: No longer, you will need to do explicit imports for pyexcel-io extensions. Instead, you install them and manage them via pip.

3.9.1 Simple Reader for a yaml file

Suppose we have a yaml file, containing a dictionary where the values are two dimensional array. The task is to write a reader plugin to pyexcel-io so that we can use get_data() to read yaml file out.

```
sheet 1:
- - 1
  - 2
  - 3
- - 2
  - 3
  - 4
sheet 2:
- - A
  - B
  - C
```

Implement IReader

First, let's implement reader interface:

1. *content_array* attribute, is expected to be a list of *NamedContent*
2. *read_sheet* function, read sheet content by its index.
3. *close* function, to clean up any file handle

```
class YourReader(IReader):
    def __init__(self, file_name, file_type, **keywords):
        self.file_handle = open(file_name, "r")
        self.native_book = yaml.load(self.file_handle)
        self.content_array = [
            NamedContent(key, values)
            for key, values in self.native_book.items()
        ]

    def read_sheet(self, sheet_index):
        two_dimensional_array = self.content_array[sheet_index].payload
        return YourSingleSheet(two_dimensional_array)

    def close(self):
        self.file_handle.close()
```

Implement ISheet

YourSingleSheet makes this simple task complex in order to show case its inner workings. Two abstract functions require implementation:

1. ***row_iterator*: should return a row: either content arary or content index as long as** *column_iterator* can use it to return the cell value.
2. *column_iterator*: should iterate cell value from the given row.

```
class YourSingleSheet(ISheet):
    def __init__(self, your_native_sheet):
        self.two_dimensional_array = your_native_sheet

    def row_iterator(self):
        yield from self.two_dimensional_array

    def column_iterator(self, row):
        yield from row
```

Plug in pyexcel-io

Last thing is to register with pyexcel-io about your new reader. *relative_plugin_class_path* meant reference from current module, how to refer to *YourReader*. *locations* meant the physical presence of the data source: "file", "memory" or "content". "file" means files on physical disk. "memory" means a file stream. "content" means a string buffer. *stream_type* meant the type of the stream: binary for BytesIO and text for StringIO.

```
IOPluginInfoChainV2(__name__).add_a_reader(
    relative_plugin_class_path="YourReader",
    locations=["file"],
    file_types=["yaml"],
    stream_type="text",
)
```

Usually, this registration code was placed in __init__.py file at the top level of your extension source tree. You can take a look at any pyexcel plugins for reference.

Test your reader

Let's run the following code and see if it works.

```
if __name__ == "__main__":
    data = get_data("test.yaml")
    print(data)
```

You would see these in standard output:

```
$ python custom_yaml_reader.py
OrderedDict([('sheet 1', [[1, 2, 3], [2, 3, 4]]), ('sheet 2', [['A', 'B', 'C']])])
```

3.9.2 A writer to write content in yaml

Now for the writer, let's write a pyexcel-io writer that write a dictionary of two dimentaional arrays back into a yaml file seen above.

Implement IWriter

Two abstract functions are required:

1. *create_sheet* creates a native sheet by sheet name, that understands how to code up the native sheet. Interestingly, it returns your sheet.
2. *close* function closes file handle if any.

```
class MyWriter(IWriter):
    def __init__(self, file_name, file_type, **keywords):
        self.file_name = file_name
```

(continues on next page)

(continued from previous page)

```
        self.content = {}

    def create_sheet(self, name):
        array = []
        self.content[name] = array
        return MySheetWriter(array)

    def close(self):
        with open(self.file_name, "w") as f:
            f.write(yaml.dump(self.content, default_flow_style=False))
```

Implement ISheetWriter

It is imagined that you will have your own sheet writer. You simply need to figure out how to write a row. Row by row write action was already written by *ISheetWrier*.

```
class MySheetWriter(ISheetWriter):
    def __init__(self, sheet_reference):
        self.native_sheet = sheet_reference

    def write_row(self, data_row):
        self.native_sheet.append(data_row)

    def close(self):
```

Plug in pyexcel-io

Like the reader plugin, we register a writer.

```
IOPluginInfoChainV2(__name__).add_a_writer(
    relative_plugin_class_path="MyWriter",
    locations=["file"],
    file_types=["yaml"],
    stream_type="text",
)
```

Test It

Let's run the following code and please examine *mytest.yaml* yourself.

```
if __name__ == "__main__":
    data_dict = {
        "sheet 1": [[1, 3, 4], [2, 4, 9]],
        "sheet 2": [["B", "C", "D"]],
    }

    save_data("mytest.yaml", data_dict)
```

And you shall find a file named 'mytest.yaml':

```
$ cat mytest.yaml
sheet 1:
- - 1
  - 3
  - 4
- - 2
  - 4
```

(continues on next page)

(continued from previous page)

```
  - 9
sheet 2:
- - B
  - C
  - D
```

3.9.3 Other pyexcel-io plugins

Get xls support

Here's what is needed:

```
>>> from pyexcel_io import save_data
>>> data = [[1,2,3]]
>>> save_data("test.xls", data)
```

And you can also get the data back:

```
>>> from pyexcel_io import get_data
>>> data = get_data("test.xls")
>>> data['pyexcel_sheet1']
[[1, 2, 3]]
```

3.9.4 Other formats

As illustrated above, you can start to play with pyexcel-xlsx, pyexcel-ods and pyexcel-ods3 plugins.

CHAPTER 4

API

4.1 Common parameters

4.1.1 'library' option is added

In order to have overlapping plugins co-exist, 'library' option is added to get_data and save_data.

4.1.2 get_data only parameters

keep_trailing_empty_cells

default: False

If turned on, the return data will contain trailing empty cells.

auto_dectect_datetime

The datetime formats are:

1. %Y-%m-%d
2. %Y-%m-%d %H:%M:%S
3. %Y-%m-%d %H:%M:%S.%f

Any other datetime formats will be thrown as ValueError

4.1.3 csv only parameters

pep_0515_off

This is related to PEP 0515, where '_' in numeric values are considered legal in python 3.6. This behavior is not consistent along with other python versions. PEP 0515 by default is suppressed. And this flag allows you to turn it on in python 3.6.

`iget_data`
`get_data`
`save_data`

CHAPTER 5

Indices and tables

- genindex
- modindex
- search