pyexcel-io Documentation

Release 0.2.0

Onni Software Ltd.

January 11, 2017

Author C.W.

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

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

License New BSD License

Version 0.2.0

Generated January 11, 2017

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.

Getting the source

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

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

CHAPTER 3

Installation

You can install it via pip:

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

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

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

Package name	Supported file formats	Dependencies	Python versions
pyexcel-io	csv, csvz [1], tsv, tsvz [2]		2.6, 2.7, 3.3, 3.4, pypy
xls	xls, xlsx(read only), xlsm(read only)	xlrd, xlwt	2.6, 2.7, 3.3, 3.4, pypy
xlsx	xlsx	openpyxl	2.6, 2.7, 3.3, 3.4, pypy
ods3	ods	ezodf, lxml	2.6, 2.7, 3.3, 3.4
ods	ods (python 2.6, 2.7)	odfpy	2.6, 2.7

Please import them before you start to access the desired file formats:

```
import pyexcel_plugin
```

After that, you can start get and save data in the loaded format.

Table 3.2: Plugin compatibility table

pyexcel-io	xls	xlsx	ods	ods3	text
0.2.0	0.2.0	0.2.0	0.2.0	0.2.0	0.2.0
0.1.0	0.1.0	0.1.0	0.1.0	0.1.0	0.1.0

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

CHAPTER 4

Special note

migration_from_dot_1_to_dot_2

CHAPTER 5

Tutorial

5.1 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,
```

5.1.1 Write to a csv file

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

```
>>> from pyexcel_io import save_data
>>> data = [[1, 2, 3], [4, 5, 6]]
>>> save_data("your_file.csv", data)
```

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')
```

5.1.2 Read from a csv file

Here's the sample code:

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

5.1.3 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
```

5.1.4 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']
>>> data = get_data(io)
>>> print(json.dumps(data))
{"csv": [["1", "2", "3"], ["4", "5", "6"]]}
```

5.2 Saving multiple sheets as CSV format

5.2.1 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)
```

5.2.2 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"]'
```

5.2.3 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
```

5.2.4 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)
>>> print(json.dumps(data))
{"Sheet 1": [["1", "2", "3"], ["4", "5", "6"]], "Sheet 2": [["7", "8", "9"], ["10", "11", "12"]]}
```

5.3 File formats: .csvz and .tsvz

5.3.1 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.

5.3.2 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"]]}'
```

5.3.3 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",
```

5.3.4 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.

5.4 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)
```

5.4.1 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.sqlbook 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)
```

Now let’s verify the data:

```
>>> from pyexcel_io.base import from_query_sets
>>> query_sets=mysession.query(Pyexcel).all()
>>> results = from_query_sets(data[0], query_sets)
>>> import json
>>> json.dumps(list(results))
'[["birth", "id", "name", "weight"], ["2014-11-11", 0, "Adam", 11.25], ["2014-11-12", 1, "Smith", 12
```

5.4.2 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.sqlbook 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
```

5.4.3 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
>>> 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()
>>> results = from_query_sets(data['Category'][0], query_sets)
>>> import json
>>> json.dumps(list(results))
'[["id", "name"], [1, "News"], [2, "Sports"]]'
>>> query_sets = mysession.query(Post).all()
>>> results = from_query_sets(["id", "title", "body", "pub_date"], query_sets)
>>> json.dumps(list(results))
'[["id", "title", "body", "pub_date"], [1, "Title A", "formal", "2015-01-20T23:28:29"], [2, "Title B
```

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

5.4.4 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", "p
```

5.5 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.

5.5.1 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.djangobook import DjangoModelImporter, DjangoModelExporter
>>> from pyexcel_io.djangobook import DjangoModelImportAdapter, DjangoModelExportAdapter
>>> model = FakeDjangoModel()
```

Suppose you have these data:

```
>>> data  = [
...     ["X", "Y", "Z"],
...     [1, 2, 3],
...     [4, 5, 6]
... ]
>>> importer = DjangoModelImporter()
>>> adapter = DjangoModelImportAdapter(model)
>>> adapter.set_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}]
```

5.5.2 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]])])
```

5.5.3 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.set_column_names(data['Sheet1'][0])
>>> adapter2 = DjangoModelImportAdapter(model2)
>>> adapter2.set_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}]
```

5.5.4 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',
```

5.6 Working with xls, xlsx, and ods formats

5.6.1 Work with physical file

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']
[[1.0, 2.0, 3.0]]
```

5.6.2 Work with memory file

Here is the sample code to work with memory file:

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

The difference is that you have mention file type if you use `pyexcel_io.save_data()`

And you can also get the data back:

```
>>> data = get_data(io, "xls")
>>> data['pyexcel_sheet1']
[[1.0, 2.0, 3.0]]
```

The same applies to `pyexcel_io.get_data()`.

5.6.3 Other formats

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

CHAPTER 6

API

Utility functions

CHAPTER 7

Indices and tables

- genindex
- modindex
- search