logo.pdf

Tablib Documentation

Release 0.11.2

February 16, 2016

Contents

I Testimonials **3**

II User's Guide **5**

1 Introduction **7**
1.1 Philosophy 7
1.2 MIT License 7
1.3 Tablib License 8
1.4 Pythons Supported 8

2 Installation **9**
2.1 Installing Tablib 9
2.2 Download the Source 9

3 Quickstart **11**
3.1 Creating a Dataset 11
3.2 Adding Rows 11
3.3 Adding Headers 12
3.4 Adding Columns 12
3.5 Importing Data 12
3.6 Exporting Data 13
3.7 Selecting Rows & Columns 13
3.8 Removing Rows & Columns 14

4 Advanced Usage **15**
4.1 Dynamic Columns 15
4.2 Filtering Datasets with Tags 16
4.3 Separators 17

5 Development **19**
5.1 Design Considerations 19

5.2 Source Control 19
5.3 Adding New Formats 20
5.4 Testing Tablib 21
5.5 Continuous Integration 22
5.6 Building the Docs 22

III API Reference **23**

6 API **25**
6.1 Dataset Object 25
6.2 Databook Object 31
6.3 Functions 32
6.4 Exceptions 32

Release v0.11.2. (*Installation*)

Tablib is an MIT Licensed format-agnostic tabular dataset library, written in Python. It allows you to import, export, and manipulate tabular data sets. Advanced features include, segregation, dynamic columns, tags & filtering, and seamless format import & export.

```
>>> data = tablib.Dataset(headers=['First Name', 'Last Name', 'Age'])
>>> map(data.append, [('Kenneth', 'Reitz', 22), ('Bessie', 'Monke', 21)])

>>> print data.json
[{"Last Name": "Reitz", "First Name": "Kenneth", "Age": 22}, {"Last Name": "Monke", "First Name"

>>> print data.yaml
- {Age: 22, First Name: Kenneth, Last Name: Reitz}
- {Age: 21, First Name: Bessie, Last Name: Monke}

>>> data.xlsx
<censored binary data>
```

Part I

TESTIMONIALS

National Geographic, Digg, Inc, Northrop Grumman, Discovery Channel, and The Sunlight Foundation use Tablib internally.

Greg Thorton Tablib by @kennethreitz saved my life. I had to consolidate like 5 huge poorly maintained lists of domains and data. It was a breeze!

Dave Coutts It's turning into one of my most used modules of 2010. You really hit a sweet spot for managing tabular data with a minimal amount of code and effort.

Joshua Ourisman Tablib has made it so much easier to deal with the inevitable 'I want an Excel file!' requests from clients...

Brad Montgomery I think you nailed the "Python Zen" with tablib. Thanks again for an awesome lib!

Part II

USER'S GUIDE

This part of the documentation, which is mostly prose, begins with some background information about Tablib, then focuses on step-by-step instructions for getting the most out of your datasets.

CHAPTER **1**

Introduction

This part of the documentation covers all the interfaces of Tablib. Tablib is a format-agnostic tabular dataset library, written in Python. It allows you to Pythonically import, export, and manipulate tabular data sets. Advanced features include, segregation, dynamic columns, tags / filtering, and seamless format import/export.

1.1 Philosophy

Tablib was developed with a few **PEP 20** idioms in mind.

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Readability counts.

All contributions to Tablib should keep these important rules in mind.

1.2 MIT License

A large number of open source projects you find today are GPL Licensed. While the GPL has its time and place, it should most certainly not be your go-to license for your next open source project.

A project that is released as GPL cannot be used in any commercial product without the product itself also being offered as open source. The MIT, BSD, and ISC licenses are great alternatives to the GPL that allow your open-source software to be used in proprietary, closed-source software.

Tablib is released under terms of The MIT License.

1.3 Tablib License

1.4 Pythons Supported

At this time, the following Python platforms are officially supported:

- cPython 2.5
- cPython 2.6
- cPython 2.7
- cPython 3.1
- cPython 3.2
- PyPy-c 1.4
- PyPy-c 1.5

Support for other Pythons will be rolled out soon.

Now, go *Install Tablib*.

CHAPTER 2

Installation

This part of the documentation covers the installation of Tablib. The first step to using any software package is getting it properly installed. Please read this section carefully, or you may miss out on some nice speed enhancements.

2.1 Installing Tablib

2.1.1 Distribute & Pip

Of course, the recommended way to install Tablib is with pip:

```
$ pip install tablib
```

2.2 Download the Source

You can also install tablib from source. The latest release (0.11.2) is available from GitHub.

- tarball
- zipball

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily.

```
$ python setup.py install
```

To download the full source history from Git, see *Source Control*.

2.2.1 Speed Extensions

New in version 0.8.5.

Tablib is partially dependent on the **pyyaml**, **simplejson**, and **xlwt** modules. To reduce installation issues, fully integrated versions of all required libraries are included in Tablib.

However, if performance is important to you (and it should be), you can install **pyyaml** with C extensions from PyPi.

```
$ pip install PyYAML
```

If you're using Python 2.5, you should also install the **simplejson** module (pip will do this for you). If you're using Python 2.6+, the built-in **json** module is already optimized and in use.

```
$ pip install simplejson
```

2.2.2 Staying Updated

The latest version of Tablib will always be available here:

- PyPi: http://pypi.python.org/pypi/tablib/
- GitHub: http://github.com/kennethreitz/tablib/

When a new version is available, upgrading is simple:

```
$ pip install tablib --upgrade
```

Now, go get a *Quick Start*.

CHAPTER 3

Quickstart

Eager to get started? This page gives a good introduction in how to get started with Tablib. This assumes you already have Tablib installed. If you do not, head over to the *Installation* section.

First, make sure that:

- Tablib is *installed*
- Tablib is *up-to-date*

Lets gets started with some simple use cases and examples.

3.1 Creating a Dataset

A `Dataset` is nothing more than what its name implies—a set of data.

Creating your own instance of the `tablib.Dataset` object is simple.

```
data = tablib.Dataset()
```

You can now start filling this `Dataset` object with data.

Example Context

From here on out, if you see `data`, assume that it's a fresh `Dataset` object.

3.2 Adding Rows

Let's say you want to collect a simple list of names.

```
# collection of names
names = ['Kenneth Reitz', 'Bessie Monke']

for name in names:
    # split name appropriately
    fname, lname = name.split()

    # add names to Dataset
    data.append([fname, lname])
```

You can get a nice, Pythonic view of the dataset at any time with `Dataset.dict`.

```
>>> data.dict
[('Kenneth', 'Reitz'), ('Bessie', 'Monke')]
```

3.3 Adding Headers

It's time to enhance our `Dataset` by giving our columns some titles. To do so, set `Dataset.headers`.

```
data.headers = ['First Name', 'Last Name']
```

Now our data looks a little different.

```
>>> data.dict
[{'Last Name': 'Reitz', 'First Name': 'Kenneth'}, {'Last Name': 'Monke', 'First Name': 'Bessie'}
```

3.4 Adding Columns

Now that we have a basic `Dataset` in place, let's add a column of **ages** to it.

```
data.append_col([22, 20], header='Age')
```

Let's view the data now.

```
>>> data.dict
[{'Last Name': 'Reitz', 'First Name': 'Kenneth', 'Age': 22}, {'Last Name': 'Monke', 'First Name'
```

It's that easy.

3.5 Importing Data

Creating a `tablib.Dataset` object by importing a pre-existing file is simple.

```
imported_data = Dataset().load(open('data.csv').read())
```

This detects what sort of data is being passed in, and uses an appropriate formatter to do the import. So you can import from a variety of different file types.

3.6 Exporting Data

Tablib's killer feature is the ability to export your `Dataset` objects into a number of formats.

Comma-Separated Values

```
>>> data.csv
Last Name,First Name,Age
Reitz,Kenneth,22
Monke,Bessie,20
```

JavaScript Object Notation

```
>>> data.json
[{"Last Name": "Reitz", "First Name": "Kenneth", "Age": 22}, {"Last Name": "Monke", "First Name"
```

YAML Ain't Markup Language

```
>>> data.yaml
- {Age: 22, First Name: Kenneth, Last Name: Reitz}
- {Age: 20, First Name: Bessie, Last Name: Monke}
```

Microsoft Excel

```
>>> data.xls
<censored binary data>
```

3.7 Selecting Rows & Columns

You can slice and dice your data, just like a standard Python list.

```
>>> data[0]
('Kenneth', 'Reitz', 22)
```

If we had a set of data consisting of thousands of rows, it could be useful to get a list of values in a column. To do so, we access the `Dataset` as if it were a standard Python dictionary.

```
>>> data['First Name']
['Kenneth', 'Bessie']
```

You can also access the column using its index.

```
>>> d.headers
['Last Name', 'First Name', 'Age']
```

```
>>> d.get_col(1)
['Kenneth', 'Bessie']
```

Let’s find the average age.

```
>>> ages = data['Age']
>>> float(sum(ages)) / len(ages)
21.0
```

3.8 Removing Rows & Columns

It’s easier than you could imagine:

```
>>> del data['Col Name']
```

```
>>> del data[0:12]
```

CHAPTER 4

Advanced Usage

This part of the documentation services to give you an idea that are otherwise hard to extract from the *API Documentation*

And now for something completely different.

4.1 Dynamic Columns

New in version 0.8.3.

Thanks to Josh Ourisman, Tablib now supports adding dynamic columns. A dynamic column is a single callable object (*ie.* a function).

Let's add a dynamic column to our `Dataset` object. In this example, we have a function that generates a random grade for our students.

```
import random

def random_grade(row):
    """Returns a random integer for entry."""
    return (random.randint(60,100)/100.0)

data.append_col(random_grade, header='Grade')
```

Let's have a look at our data.

```
>>> data.yaml
- {Age: 22, First Name: Kenneth, Grade: 0.6, Last Name: Reitz}
- {Age: 20, First Name: Bessie, Grade: 0.75, Last Name: Monke}
```

Let's remove that column.

```
>>> del data['Grade']
```

When you add a dynamic column, the first argument that is passed in to the given callable is the current data row. You can use this to perform calculations against your data row.

For example, we can use the data available in the row to guess the gender of a student.

```
def guess_gender(row):
    """Calculates gender of given student data row."""
    m_names = ('Kenneth', 'Mike', 'Yuri')
    f_names = ('Bessie', 'Samantha', 'Heather')

    name = row[0]

    if name in m_names:
        return 'Male'
    elif name in f_names:
        return 'Female'
    else:
        return 'Unknown'
```

Adding this function to our dataset as a dynamic column would result in:

```
>>> data.yaml
- {Age: 22, First Name: Kenneth, Gender: Male, Last Name: Reitz}
- {Age: 20, First Name: Bessie, Gender: Female, Last Name: Monke}
```

4.2 Filtering Datasets with Tags

New in version 0.9.0.

When constructing a `Dataset` object, you can add tags to rows by specifying the `tags` parameter. This allows you to filter your `Dataset` later. This can be useful to separate rows of data based on arbitrary criteria (*e.g.* origin) that you don't want to include in your `Dataset`.

Let's tag some students.

```
students = tablib.Dataset()

students.headers = ['first', 'last']

students.rpush(['Kenneth', 'Reitz'], tags=['male', 'technical'])
students.rpush(['Bessie', 'Monke'], tags=['female', 'creative'])
```

Now that we have extra meta-data on our rows, we can easily filter our `Dataset`. Let's just see Male students.

```
>>> students.filter(['male']).yaml
- {first: Kenneth, Last: Reitz}
```

It's that simple. The original `Dataset` is untouched.

4.2.1 Excel Workbook With Multiple Sheets

When dealing with a large number of `Datasets` in spreadsheet format, it's quite common to group multiple spreadsheets into a single Excel file, known as a Workbook. Tablib makes it extremely easy to build workbooks with the handy, `Databook` class.

Let's say we have 3 different `Datasets`. All we have to do is add then to a `Databook` object...

```
book = tablib.Databook((data1, data2, data3))
```

... and export to Excel just like `Datasets`.

```
with open('students.xls', 'wb') as f:
    f.write(book.xls)
```

The resulting **students.xls** file will contain a separate spreadsheet for each `Dataset` object in the `Databook`.

Binary Warning

Make sure to open the output file in binary mode.

4.3 Separators

New in version 0.8.2.

When, it's often useful to create a blank row containing information on the upcoming data. So,

```
daniel_tests = [
    ('11/24/09', 'Math 101 Mid-term Exam', 56.),
    ('05/24/10', 'Math 101 Final Exam', 62.)
]

suzie_tests = [
    ('11/24/09', 'Math 101 Mid-term Exam', 56.),
    ('05/24/10', 'Math 101 Final Exam', 62.)
]

# Create new dataset
tests = tablib.Dataset()
tests.headers = ['Date', 'Test Name', 'Grade']

# Daniel's Tests
tests.append_separator('Daniel\'s Scores')

for test_row in daniel_tests:
   tests.append(test_row)
```

```
# Susie's Tests
tests.append_separator('Susie\'s Scores')

for test_row in suzie_tests:
   tests.append(test_row)

# Write spreadsheet to disk
with open('grades.xls', 'wb') as f:
    f.write(tests.xls)
```

The resulting **tests.xls** will have the following layout:

Daniel's Scores:

- '11/24/09', 'Math 101 Mid-term Exam', 56.
- '05/24/10', 'Math 101 Final Exam', 62.

Suzie's Scores:

- '11/24/09', 'Math 101 Mid-term Exam', 56.
- '05/24/10', 'Math 101 Final Exam', 62.

Format Support

At this time, only `Excel` output supports separators.

Now, go check out the *API Documentation* or begin *Tablib Development*.

CHAPTER 5

Development

Tablib is under active development, and contributors are welcome.

If you have a feature request, suggestion, or bug report, please open a new issue on GitHub. To submit patches, please send a pull request on GitHub.

5.1 Design Considerations

Tablib was developed with a few **PEP 20** idioms in mind.

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Readability counts.

A few other things to keep in mind:

1. Keep your code DRY.
2. Strive to be as simple (to use) as possible.

5.2 Source Control

Tablib source is controlled with Git, the lean, mean, distributed source control machine.

The repository is publicly accessible.

```
git clone git://github.com/kennethreitz/tablib.git
```

The project is hosted on **GitHub**.

GitHub: http://github.com/kennethreitz/tablib

5.2.1 Git Branch Structure

Feature / Hotfix / Release branches follow a Successful Git Branching Model . Git-flow is a great tool for managing the repository. I highly recommend it.

`develop` The "next release" branch. Likely unstable.

`master` Current production release (0.11.2) on PyPi.

Each release is tagged.

When submitting patches, please place your feature/change in its own branch prior to opening a pull request on GitHub.

5.3 Adding New Formats

Tablib welcomes new format additions! Format suggestions include:

- MySQL Dump

5.3.1 Coding by Convention

Tablib features a micro-framework for adding format support. The easiest way to understand it is to use it. So, let's define our own format, named *xxx*.

1. Write a new format interface.

   `tablib.core` follows a simple pattern for automatically utilizing your format throughout Tablib. Function names are crucial.

   Example **tablib/formats/_xxx.py**:

```
title = 'xxx'

def export_set(dset):
    ....
    # returns string representation of given dataset

def export_book(dbook):
    ....
    # returns string representation of given databook

def import_set(dset, in_stream):
    ...
    # populates given Dataset with given datastream

def import_book(dbook, in_stream):
    ...
```

```
            # returns Databook instance

        def detect(stream):
            ...
            # returns True if given stream is parsable as xxx
```

Excluding Support

If the format excludes support for an import/export mechanism (*eg.* `csv` excludes `Databook` support), simply don't define the respective functions. Appropriate errors will be raised.

---

2. Add your new format module to the `tablib.formats.available` tuple.
3. Add a mock property to the `Dataset` class with verbose reStructured Text docstring. This alleviates IDE confusion, and allows for pretty auto-generated Sphinx documentation.
4. Write respective *tests*.

5.4 Testing Tablib

Testing is crucial to Tablib's stability. This stable project is used in production by many companies and developers, so it is important to be certain that every version released is fully operational. When developing a new feature for Tablib, be sure to write proper tests for it as well.

When developing a feature for Tablib, the easiest way to test your changes for potential issues is to simply run the test suite directly.

```
$ ./test_tablib.py
```

Jenkins CI, amongst other tools, supports Java's xUnit testing report format. Nose allows us to generate our own xUnit reports.

Installing nose is simple.

```
$ pip install nose
```

Once installed, we can generate our xUnit report with a single command.

```
$ nosetests test_tablib.py --with-xunit
```

This will generate a **nosetests.xml** file, which can then be analyzed.

5.5 Continuous Integration

Every commit made to the **develop** branch is automatically tested and inspected upon receipt with **'Travis CI'_**. If you have access to the main repository and broke the build, you will receive an email accordingly.

Anyone may view the build status and history at any time.

> https://travis-ci.org/kennethreitz/tablib

Additional reports will also be included here in the future, including **PEP 8** checks and stress reports for extremely large datasets.

5.6 Building the Docs

Documentation is written in the powerful, flexible, and standard Python documentation format, reStructured Text. Documentation builds are powered by the powerful Pocoo project, Sphinx. The *API Documentation* is mostly documented inline throughout the module.

The Docs live in `tablib/docs`. In order to build them, you will first need to install Sphinx.

```
$ pip install sphinx
```

Then, to build an HTML version of the docs, simply run the following from the **docs** directory:

```
$ make html
```

Your `docs/_build/html` directory will then contain an HTML representation of the documentation, ready for publication on most web servers.

You can also generate the documentation in **epub**, **latex**, **json**, *&c* similarly.

Make sure to check out the *API Documentation*.

Part III

API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

CHAPTER **6**

API

This part of the documentation covers all the interfaces of Tablib. For parts where Tablib depends on external libraries, we document the most important right here and provide links to the canonical documentation.

6.1 Dataset Object

class `tablib.`**`Dataset`**(**args*, ***kwargs*)

The `Dataset` object is the heart of Tablib. It provides all core functionality.

Usually you create a `Dataset` instance in your main module, and append rows as you collect data.

```
data = tablib.Dataset()
data.headers = ('name', 'age')

for (name, age) in some_collector():
    data.append((name, age))
```

Setting columns is similar. The column data length must equal the current height of the data and headers must be set

```
data = tablib.Dataset()
data.headers = ('first_name', 'last_name')

data.append(('John', 'Adams'))
data.append(('George', 'Washington'))

data.append_col((90, 67), header='age')
```

You can also set rows and headers upon instantiation. This is useful if dealing with dozens or hundreds of `Dataset` objects.

```
headers = ('first_name', 'last_name')
data = [('John', 'Adams'), ('George', 'Washington')]
data = tablib.Dataset(*data, headers=headers)
```

Parameters

- ***args** – (optional) list of rows to populate Dataset
- **headers** – (optional) list strings for Dataset header row

---

Format Attributes Definition

If you look at the code, the various output/import formats are not defined within the `Dataset` object. To add support for a new format, see *Adding New Formats*.

---

add_formatter(*col, handler*)
Adds a formatter to the `Dataset`.

New in version 0.9.5: :param col: column to. Accepts index int or header str. :param handler: reference to callback function to execute against each cell value.

append(*row, tags=[]*)
Adds a row to the `Dataset`. See `Dataset.insert` for additional documentation.

append_col(*col, header=None*)
Adds a column to the `Dataset`. See `Dataset.insert_col` for additional documentation.

append_separator(*text='-'*)
Adds a *separator* to the `Dataset`.

csv
A CSV representation of the `Dataset` object. The top row will contain headers, if they have been set. Otherwise, the top row will contain the first row of the dataset.

A dataset object can also be imported by setting the `Dataset.csv` attribute.

```
data = tablib.Dataset()
data.csv = 'age, first_name, last_name\n90, John, Adams'
```

Import assumes (for now) that headers exist.

Binary Warning

`Dataset.csv` uses rn line endings by default, so make sure to write in binary mode:

```
with open('output.csv', 'wb') as f:
    f.write(data.csv)
```

If you do not do this, and you export the file on Windows, your CSV file will open in Excel with a blank line between each row.

dbf

A dBASE representation of the `Dataset` object.

A dataset object can also be imported by setting the `Dataset.dbf` attribute.

```
# To import data from an existing DBF file:
data = tablib.Dataset()
data.dbf = open('existing_table.dbf').read()

# to import data from an ASCII-encoded bytestring:
data = tablib.Dataset()
data.dbf = '<bytestring of tabular data>'
```

Binary Warning

`Dataset.dbf` contains binary data, so make sure to write in binary mode:

```
with open('output.dbf', 'wb') as f:
    f.write(data.dbf)
```

dict

A native Python representation of the `Dataset` object. If headers have been set, a list of Python dictionaries will be returned. If no headers have been set, a list of tuples (rows) will be returned instead.

A dataset object can also be imported by setting the *Dataset.dict* attribute:

```
data = tablib.Dataset()
data.dict = [{'age': 90, 'first_name': 'Kenneth', 'last_name': 'Reitz'}]
```

export(*format*, ***kwargs*)

Export `Dataset` object to *format*.

Parameters ****kwargs** – (optional) custom configuration to the format *export_set*.

extend(*rows*, *tags=[]*)

Adds a list of rows to the `Dataset` using `Dataset.append`

filter(*tag*)

Returns a new instance of the `Dataset`, excluding any rows that do not contain the given *tags*.

get_col(*index*)

Returns the column from the `Dataset` at the given index.

headers

An *optional* list of strings to be used for header rows and attribute names.

This must be set manually. The given list length must equal `Dataset.width`.

`height`
The number of rows currently in the `Dataset`. Cannot be directly modified.

`html`
A HTML table representation of the `Dataset` object. If headers have been set, they will be used as table headers.

..notice:: This method can be used for export only.

`insert`(*index*, *row*, *tags=[]*)
Inserts a row to the `Dataset` at the given index.

Rows inserted must be the correct size (height or width).

The default behaviour is to insert the given row to the `Dataset` object at the given index.

`insert_col`(*index*, *col=None*, *header=None*)
Inserts a column to the `Dataset` at the given index.

Columns inserted must be the correct height.

You can also insert a column of a single callable object, which will add a new column with the return values of the callable each as an item in the column.

```
data.append_col(col=random.randint)
```

If inserting a column, and `Dataset.headers` is set, the header attribute must be set, and will be considered the header for that row.

See *Dynamic Columns* for an in-depth example.

Changed in version 0.9.0: If inserting a column, and `Dataset.headers` is set, the header attribute must be set, and will be considered the header for that row.

New in version 0.9.0: If inserting a row, you can add *tags* to the row you are inserting. This gives you the ability to `filter` your `Dataset` later.

`insert_separator`(*index*, *text='-'*)
Adds a separator to `Dataset` at given index.

`json`
A JSON representation of the `Dataset` object. If headers have been set, a JSON list of objects will be returned. If no headers have been set, a JSON list of lists (rows) will be returned instead.

A dataset object can also be imported by setting the `Dataset.json` attribute:

```
data = tablib.Dataset()
data.json = '[{"age": 90, "first_name": "John", "last_name": "Adams"}]'
```

Import assumes (for now) that headers exist.

`latex`
A LaTeX booktabs representation of the `Dataset` object. If a title has been set, it will be exported as the table caption.

Note: This method can be used for export only.

`load`(*in_stream*, *format=None*, ***kwargs*)
Import *in_stream* to the `Dataset` object using the *format*.

Parameters **`**kwargs`** – (optional) custom configuration to the format *import_set*.

`lpop`()
Removes and returns the first row of the `Dataset`.

`lpush`(*row*, *tags=[]*)
Adds a row to the top of the `Dataset`. See `Dataset.insert` for additional documentation.

`lpush_col`(*col*, *header=None*)
Adds a column to the top of the `Dataset`. See `Dataset.insert` for additional documentation.

`ods`
An OpenDocument Spreadsheet representation of the `Dataset` object, with *Separators*. Cannot be set.

Binary Warning

`Dataset.ods` contains binary data, so make sure to write in binary mode:

```
with open('output.ods', 'wb') as f:
    f.write(data.ods)
```

`pop`()
Removes and returns the last row of the `Dataset`.

`remove_duplicates`()
Removes all duplicate rows from the `Dataset` object while maintaining the original order.

`rpop`()
Removes and returns the last row of the `Dataset`.

`rpush`(*row*, *tags=[]*)
Adds a row to the end of the `Dataset`. See `Dataset.insert` for additional documentation.

`rpush_col`(*col*, *header=None*)
Adds a column to the end of the `Dataset`. See `Dataset.insert` for additional documentation.

sort(*col*, *reverse=False*)

Sort a `Dataset` by a specific column, given string (for header) or integer (for column index). The order can be reversed by setting `reverse` to `True`.

Returns a new `Dataset` instance where columns have been sorted.

stack(*other*)

Stack two `Dataset` instances together by joining at the row level, and return new combined `Dataset` instance.

stack_cols(*other*)

Stack two `Dataset` instances together by joining at the column level, and return a new combined `Dataset` instance. If either `Dataset` has headers set, than the other must as well.

subset(*rows=None*, *cols=None*)

Returns a new instance of the `Dataset`, including only specified rows and columns.

transpose()

Transpose a `Dataset`, turning rows into columns and vice versa, returning a new `Dataset` instance. The first row of the original instance becomes the new header row.

tsv

A TSV representation of the `Dataset` object. The top row will contain headers, if they have been set. Otherwise, the top row will contain the first row of the dataset.

A dataset object can also be imported by setting the `Dataset.tsv` attribute.

```
data = tablib.Dataset()
data.tsv = 'age     first_name      last_name\n90   John     Adams'
```

Import assumes (for now) that headers exist.

width

The number of columns currently in the `Dataset`. Cannot be directly modified.

wipe()

Removes all content and headers from the `Dataset` object.

xls

A Legacy Excel Spreadsheet representation of the `Dataset` object, with *Separators*. Cannot be set.

Note:

XLS files are limited to a maximum of 65,000 rows. Use `Dataset.xlsx` to avoid this limitation.

Binary Warning

`Dataset.xls` contains binary data, so make sure to write in binary mode:

```
with open('output.xls', 'wb') as f:
    f.write(data.xls)
```

`xlsx`

An Excel '07+ Spreadsheet representation of the `Dataset` object, with *Separators*. Cannot be set.

Binary Warning

`Dataset.xlsx` contains binary data, so make sure to write in binary mode:

```
with open('output.xlsx', 'wb') as f:
    f.write(data.xlsx)
```

`yaml`

A YAML representation of the `Dataset` object. If headers have been set, a YAML list of objects will be returned. If no headers have been set, a YAML list of lists (rows) will be returned instead.

A dataset object can also be imported by setting the `Dataset.yaml` attribute:

```
data = tablib.Dataset()
data.yaml = '- {age: 90, first_name: John, last_name: Adams}'
```

Import assumes (for now) that headers exist.

6.2 Databook Object

class `tablib.`**`Databook`**(*sets=None*)

A book of `Dataset` objects.

`add_sheet`(*dataset*)

Adds given `Dataset` to the `Databook`.

`export`(*format*, ***kwargs*)

Export `Databook` object to *format*.

Parameters **`**kwargs`** – (optional) custom configuration to the format *export_book*.

`load`(*format*, *in_stream*, ***kwargs*)

Import *in_stream* to the `Databook` object using the *format*.

Parameters **`**kwargs`** – (optional) custom configuration to the format *import_book*.

size
The number of the `Dataset` objects within `Databook`.

wipe()
Removes all `Dataset` objects from the `Databook`.

6.3 Functions

`tablib.`**`import_set`**(*stream*, *format=None*, ***kwargs*)
Return dataset of given stream.

6.4 Exceptions

class `tablib.`**`InvalidDatasetType`**
You're trying to add something that doesn't quite look right.

class `tablib.`**`InvalidDimensions`**
You're trying to add something that doesn't quite fit right.

class `tablib.`**`UnsupportedFormat`**
You're trying to add something that doesn't quite taste right.

Now, go start some *Tablib Development*.

Index

A

add_formatter() (tablib.Dataset method), 26
add_sheet() (tablib.Databook method), 31
append() (tablib.Dataset method), 26
append_col() (tablib.Dataset method), 26
append_separator() (tablib.Dataset method), 26

C

csv (tablib.Dataset attribute), 26

D

Databook (class in tablib), 31
Dataset (class in tablib), 25
dbf (tablib.Dataset attribute), 27
dict (tablib.Dataset attribute), 27

E

export() (tablib.Databook method), 31
export() (tablib.Dataset method), 27
extend() (tablib.Dataset method), 27

F

filter() (tablib.Dataset method), 27

G

get_col() (tablib.Dataset method), 27

H

headers (tablib.Dataset attribute), 27
height (tablib.Dataset attribute), 28
html (tablib.Dataset attribute), 28

I

import_set() (in module tablib), 32
insert() (tablib.Dataset method), 28
insert_col() (tablib.Dataset method), 28
insert_separator() (tablib.Dataset method), 28
InvalidDatasetType (class in tablib), 32
InvalidDimensions (class in tablib), 32

J

json (tablib.Dataset attribute), 28

L

latex (tablib.Dataset attribute), 28
load() (tablib.Databook method), 31
load() (tablib.Dataset method), 29
lpop() (tablib.Dataset method), 29
lpush() (tablib.Dataset method), 29
lpush_col() (tablib.Dataset method), 29

O

ods (tablib.Dataset attribute), 29

P

pop() (tablib.Dataset method), 29
Python Enhancement Proposals
    PEP 20, 7, 19
    PEP 8, 22

R

remove_duplicates() (tablib.Dataset method), 29
rpop() (tablib.Dataset method), 29
rpush() (tablib.Dataset method), 29
rpush_col() (tablib.Dataset method), 29

S

size (tablib.Databook attribute), 31

sort() (tablib.Dataset method), 29
stack() (tablib.Dataset method), 30
stack_cols() (tablib.Dataset method), 30
subset() (tablib.Dataset method), 30

T

tablib (module), 11, 25
transpose() (tablib.Dataset method), 30
tsv (tablib.Dataset attribute), 30

U

UnsupportedFormat (class in tablib), 32

W

width (tablib.Dataset attribute), 30
wipe() (tablib.Databook method), 32
wipe() (tablib.Dataset method), 30

X

xls (tablib.Dataset attribute), 30
xlsx (tablib.Dataset attribute), 31

Y

yaml (tablib.Dataset attribute), 31