tabel Documentation

Release v1.0

Bastiaan Bergman

May 20, 2019

Contents:

Tabel provides a lightweight and intuitive object for data tables, two dimensional tables with rows and columns. Based on Numpy, therefore fast and seamless integration with all the popular data tools.

CHAPTER 1

About Tabel

Lightweight, intuitive and fast data-tables.

Tabel data-tables are tables with columns and column names, rows and row numbers. Indexing and slicing your data is analogous to numpy array's. The only real difference is that each column can have its own data type.

1.1 Design objectives

I got frustrated with pandas: it's complicated slicing syntax (.loc, .x, .iloc, .. etc), it's enforced index column and the Series objects I get when I want a numpy array. With Tabel I created the simplified pandas I need for many of my data-jobs. Just focussing on simple slicing of multi-datatype tables and basic table tools.

- Intuitive simple slicing.
- Using numpy machinery, for best performance, integration with other tools and future support.
- Store data by column numpy arrays (column store).
- No particular index column, all columns can be used as the index, the choice is up to the user.
- Fundamental necessities for sorting, grouping, joining and appending tables.

CHAPTER 2

Install

pip install tabel

CHAPTER 3

Quickstart

3.1 init

To setup a Tabel:

```
>>> from tabel import Tabel
>>> tbl = Tabel([ ["John", "Joe", "Jane"],
...                [1.82,1.65,2.15],
...                [False,False,True]], columns = ["Name", "Height", "Married"])
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 John   |     1.82 |         0
 Joe    |     1.65 |         0
 Jane   |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
```

Alternatively, Tabels can be setup from dictionaries, numpy arrays, pandas DataFrames, or no data at all. Database connectors usually return data as a list of records, the module provides a convenience function to transpose this into a list of columns.

3.2 slice

Slicing can be done the numpy way, always returning Tabel objects:

```
>>> tbl[1:3,[0,2]]
 Name   |   Married
--------+-----------
 Joe    |         0
 Jane   |         1
2 rows ['<U4', '|b1']
```

Slices will always return a Tabel except in three distinct cases, when:

1. explicitly one column is requested, a numpy array is returned:

```
>>> tbl[1:3,'Name']        # doctest: +SKIP
array(['Joe', 'Jane'],
      dtype='<U4')
```

2. explicitly one row is requested, a tuple is returned:

```
>>> tbl[0,:]
('John', 1.82, False)
```

3. explicitly one element is requested:

```
>>> tbl[0,'Name']
'John'
```

In general, slicing is intuitive and does not deviate from what would expect from numpy. With the one addition that columns can be referred to by names as well as numbers.

3.3 set

Setting elements works the same as slicing:

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.15],
↪'Married': [False,False,True]})
>>> tbl[0,"Name"] = "Jos"
>>> tbl
 Name    |   Height |   Married
---------+----------+-----------
 Jos     |     1.82 |         0
 Joe     |     1.65 |         0
 Jane    |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
```

The datatype that the value is expected to have, is the same as the datatype a slice would result into.

Adding columns, works the same as setting elements, just give it a new name:

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.15],
↪'Married': [False,False,True]})
>>> tbl['new'] = [1,2,3]
>>> tbl
 Name    |   Height |   Married |   new
---------+----------+-----------+-------
 John    |     1.82 |         0 |     1
 Joe     |     1.65 |         0 |     2
 Jane    |     2.15 |         1 |     3
3 rows ['<U4', '<f8', '|b1', '<i8']
```

Or set the whole column to the same value:

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.15],
↪'Married': [False,False,True]})
>>> tbl['new'] = 13
```

(continues on next page)

(continued from previous page)

```
>>> tbl
 Name    |   Height |   Married |   new
---------+----------+-----------+-------
 John    |     1.82 |         0 |    13
 Joe     |     1.65 |         0 |    13
 Jane    |     2.15 |         1 |    13
3 rows ['<U4', '<f8', '|b1', '<i8']
```

Just like numpy, slices are not actual copies of the data, rather they are references.

3.4 append Tabel and row

Tabels can be appended with other Tabels:

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.15],
↪'Married': [False,False,True]})
>>> tbl += tbl
>>> tbl
 Name    |   Height |   Married
---------+----------+-----------
 John    |     1.82 |         0
 Joe     |     1.65 |         0
 Jane    |     2.15 |         1
 John    |     1.82 |         0
 Joe     |     1.65 |         0
 Jane    |     2.15 |         1
6 rows ['<U4', '<f8', '|b1']
```

Or append rows as dictionary:

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.15],
↪'Married': [False,False,True]})
>>> tbl.row_append({'Height':1.81, 'Name':"Jack", 'Married':True})
>>> tbl
 Name    |   Height |   Married
---------+----------+-----------
 John    |     1.82 |         0
 Joe     |     1.65 |         0
 Jane    |     2.15 |         1
 Jack    |     1.81 |         1
4 rows ['<U4', '<f8', '|b1']
```

3.5 instance properties

Your data is simply stored as a list of numpy arrays and can be accessed or manipulated like that (just don't make a mess):

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.15],
↪'Married': [False,False,True]})
>>> tbl.columns
['Name', 'Height', 'Married']
>>> tbl.data        # doctest: +SKIP
```

(continues on next page)

(continued from previous page)

```
[array(['John', 'Joe', 'Jane'],
      dtype='<U4'), array([ 1.82,  1.65,  2.15]), array([False, False,  True],␣
↪dtype=bool)]
```

Further the basic means to asses the size of your data:

```
>>> tbl.shape
(3, 3)
>>> len(tbl)
3
```

3.6 pandas

For for interfacing with the popular datatable framework, going back and forth is easy:

```
>>> import pandas as pd
>>> df = pd.DataFrame({'a':range(3),'b':range(10,13)})
>>> df
   a   b
0  0  10
1  1  11
2  2  12
```

To make a Tabel from a DataFrame, just supply it to the initialize:

```
>>> tbl = Tabel(df)
>>> tbl
   a |   b
-----+-----
   0 |  10
   1 |  11
   2 |  12
3 rows ['<i8', '<i8']
```

The dict property of Tabel provides a way to make a DataFrame from a Tabel:

```
>>> df = pd.DataFrame(tbl.dict)
>>> df
   a   b
0  0  10
1  1  11
2  2  12
```

CHAPTER 4

Resources & getting help

- See for the full API and more examples the documentation on RTD.
- The repository on Github.
- Installables on pip.
- Questions and answers on StackOverflow, I will try to monitor for it.

CHAPTER 5

Stable releases

- tabel 1.2.1
  - Removed unicode characters from description to fix pip install *issue <https://github.com/BastiaanBergman/tabel/issues/6#issue-440282452>*.
- tabel 1.2.0
  - Fix for numpy 1.15.5 “warnings”
  - Fix for outerjoin to raise an error in case of unsupported datatypes
- tabel 1.1
  - Added join and group_by methods
  - September 27, 2018
- tabel 1.0
  - First release
  - September 8, 2018

CHAPTER 6

Dependencies

- numpy
- tabulate (optional, recommended)
- pandas (optional, for converting back and forth to DataFrames)

6.1 Tested on:

- Python 3.6.4; numpy 1.15.4
- Python 3.6.4; numpy 1.14.3
- Python 2.7.14; numpy 1.14.0

CHAPTER 7

Contributing to Tabel

Tabel is perfect already, no more contributions needed. Just kidding!

See the repository for filing issues and proposing enhancements.

I'm using pytest, pylint, doctest, sphynx and setuptools.

- pytest

```
cd tabel/test
conda activate py3_6
pytest
conda activate py2_7
pytest
```

- pylint

```
cd tabel/
./pylint.sh
```

- doctest

```
cd tabel/docs
make doctest
```

- sphynx

```
cd tabel/docs
make html
```

- setuptools/pypi

```
python setup.py sdist bdist_wheel
twine upload dist/tabel-1.1.0.*
```

CHAPTER 8

Contributors

Just me, Bastiaan Bergman [Bastiaan.Bergman@gmail.com].

CHAPTER 9

What's in the name?

Tabel is Dutch for table (two-dimensional enlisting), wiktionary. The english word table, as in "dinner table", translates in Dutch to *tafel*. The Dutch word *tafel* is an old fashioned word for data-table, mostly used for calculation tables which itself is old fashioned as well.

CHAPTER 10

ToDo

- polish error messages and validity checking and add testing for it.
- cache buffers for faster appending: store temp in list and concatenate to array only once we use another method
- allow for (sparse) numpy arrays as an element
- adjust & limit __repr__ width for very wide Tabels in jupyter cell
- items() and row_items() and keys() and values() method
- pop_column method
- tox - environment testing
- set subsets of tabels with (subsets) of other tabels, seems logic as __setitem__ is allowed to provide the datatype that should have come from a __getitem__
- datetime column support
- add disk datalogger

CHAPTER 11

Performance

Tabel is designed for those cases where you have data of multiple types, like records in a database, where the volume is small enough to fit it all in memory and high freedom of data manipulation is needed. Use Tabel when the following conditions are met:

- Multiple columns with different datatyes each
- Fields in a row have a shared relation (e.g. records in a database)
- All data fits in memory
- Only few and relatively simple SQL queries are needed
- A large proportion of the data is numeric and would need column-wise arithmetic
- Data manipulation requires complex (computed) slicing across both dimensions (rows and columns)

If the above conditions are not met, consider these alternatives:

- numpy - if your data is purely numerical or when fields in a row have no relation.
- HDF5 - if your data does not fit in memory
- SQLite or MySQL - if you need complex DB queries (e.g. transactional)

That said, good general performance is paramount. Here below are some tests to give an rough overview of where Tabel stands relative to pandas.

To do the tests, a few tools are needed:

```
>>> import numpy as np
>>> import pandas as pd
>>> from tabel import Tabel
>>> from timeit import default_timer
```

11.1 slicing tables

As Tabel is based on numpy, this is where it is good at. I think slicing syntax is both clean and fast (about 15x faster than pandas in the below tests).

First slicing test is to get a whole row from a large table. The return value should be a tuple from Tabel, pandas produces an object array which I believe is not very efficient as columns usually have different datatypes. Tabel is abut 43x faster in this particular case:

```
>>> n = 1000000
>>> data_dict = {'a':[1,2,3,4] * n, 'b':['a1','a2']*2*n, 'c':np.arange(4*n)}
>>> df = pd.DataFrame(data_dict)
>>> tbl = Tabel(data_dict)
>>> def test_pandas_slice(n):
...     t0 = default_timer()
...     for i in range(n):
...         _ = df.iloc[100].values
...     return (default_timer() - t0)/n*1e6, "micro-sec"
>>> test_pandas_slice(100000)                    # doctest: +SKIP
(130.62096348032355, 'micro-sec')
>>> def test_tabel_slice(n):
...     t0 = default_timer()
...     for i in range(n):
...         _ = tbl[100]
...     return (default_timer() - t0)/n*1e6, "micro-sec"
>>> test_tabel_slice(100000)                     # doctest: +SKIP
(3.045091559179127, 'micro-sec')
```

Second slicing test is to get a number of rows from one particular column, the return value should be an array. Tabel is abut 15x faster in this case:

```
>>> def test_pandas_slice(n):
...     t0 = default_timer()
...     for i in range(n):
...         _ = df['a'][1:10].values
...     return (default_timer() - t0)/n*1e6, "micro-sec"
>>> test_pandas_slice(100000)                    # doctest: +SKIP
(63.41531826881692, 'micro-sec')
>>> def test_tabel_slice(n):
...     t0 = default_timer()
...     for i in range(n):
...         _ = tbl[1:10, 'a']
...     return (default_timer() - t0)/n*1e6, "micro-sec"
>>> test_tabel_slice(100000)                     # doctest: +SKIP
(4.007692479062825, 'micro-sec')
```

11.2 appending rows

I find myself often doing something similar to the following procedure:

```
>>> for i in large_number:                       # doctest: +SKIP
...   row = calculate_something()
...   tbl.append(row)
```

where "do something" could be reading some tricky formatted ascii file or calculating the position of the stars, it

doesn't matter. The point is, appending data is never a very efficient thing to do, memory needs to be reserved, all values need to be re-evaluated for type setting and then all data gets copied over to the new location. Still I prefer this over more efficient methods as it keeps my code clean and simple.

I expect bad performance for both Tabel and pandas, but a simple test shows that pandas is doing a bit worse. If we repeatedly add a simple row of two integers to a pandas DataFrame we'll use about 300 micro-second per row:

```
>>> import pandas as pd
>>> from tabel import Tabel
>>> from timeit import default_timer
>>> df = pd.DataFrame({'1':[],'2':[]})
>>> row = pd.DataFrame({'1':[1],'2':[2]})
>>> def test_pandas(n):
...     t0 = default_timer()
...     for i in range(n):
...         df.append(row, ignore_index=True)
...     return (default_timer() - t0)/n*1e6, "micro-sec"
>>> test_pandas(100000)                              # doctest: +SKIP
(299.0699630905874, 'micro-sec')
```

The same exercise with Tabel's *tabel.Tabel.row_append* method takes only a quarter of that time:

```
>>> tbl = Tabel()
>>> row = (1, 2)
>>> def test_tabel(n):
...     t0 = default_timer()
...     for i in range(n):
...         tbl.row_append(row)
...     return (default_timer() - t0)/n*1e6, "micro-sec"
>>> test_tabel(100000)                               # doctest: +SKIP
(79.83603572938591, 'micro-sec')
```

Granted, there are very many different scenarios thinkable and there probably are scenarios where pandas would outperform Tabel. If you come across one of those please let me know and I happily add it here.

11.3 grouping tables

A big Tabel with a million rows can be grouped by multiple columns. Both pandas and Tabel take a good amount of time on this, but then this is typically done once on a an individual Tabel or DataFrame. pandas is about 10x faster than Tabel on this simple test. The good news is that this is independent of n, meaning they're both equally scalable.

```
>>> n = 100000
>>> data_dict = {'a':[1,2,3,4] * n, 'b':['a1','a2']*2*n, 'c':np.arange(4*n)}
>>> df = pd.DataFrame(data_dict)
>>> def test_pandas_groupby(n):
...     t0 = default_timer()
...     for i in range(n):
...         _ = df.groupby(('a', 'b')).sum()
...     return (default_timer() - t0)/n*1e3, "mili-sec"
>>> test_pandas_groupby(10)                          # doctest: +SKIP
(34.32465291116387, 'mili-sec')
>>> tbl = Tabel(data_dict)
>>> def test_tabel_groupby(n):
...     t0 = default_timer()
...     for i in range(n):
...         _ = tbl.group_by(('b', 'a'),[(np.sum, 'c')])
```

(continues on next page)

(continued from previous page)

```
...     return (default_timer() - t0)/n*1e3, "mili-sec"
>>> test_tabel_groupby(10)                          # doctest: +SKIP
(322.54316059406847, 'mili-sec')
```

11.4 joining tables

Two Tabels can be joined together by some common key present in both Tabels. Two table with a million rows takes about 2.5 second to be joined with pandas and 5 times that with Tabel, this ratio reduces somewhat with n. See the codeblock below for the specific case tested here.

```
>>> n = 1000000
>>> data_dict = {'a':[1,2,3,4] * n, 'b':['a1','a2']*2*n, 'c':np.arange(4*n)}
>>> df_1 = pd.DataFrame(data_dict)
>>> df_2 = pd.DataFrame(data_dict)
>>> def test_pandas_join():
...     t0 = default_timer()
...     _ = df_1.join(df_2, on='c', how='inner', lsuffix='l', rsuffix='r')
...     return (default_timer() - t0)*1e3, "mili-sec"
>>> test_pandas_join()                              # doctest: +SKIP
(2462.8482228145003, 'mili-sec')
>>> tbl_1 = Tabel(data_dict)
>>> tbl_2 = Tabel(data_dict)
>>> def test_tabel_join():
...     t0 = default_timer()
...     _ = tbl_1.join(tbl_2, key='c', jointype='inner')
...     return (default_timer() - t0)*1e3, "mili-sec"
>>> test_tabel_join()                               # doctest: +SKIP
(10393.40596087277, 'mili-sec')
```

CHAPTER 12

Cookbook

12.1 Initialization

API documentation: *tabel.Tabel*.

From a list of lists and a separate list of column names:

```
>>> from tabel import Tabel
>>> tbl = Tabel([ ["John", "Joe", "Jane"],
...                [1.82,1.65,2.15],
...                [False,False,True]], columns = ["Name", "Height", "Married"])
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 John   |     1.82 |         0
 Joe    |     1.65 |         0
 Jane   |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
```

From a dictionary:

```
>>> data = {'Name' : ["John", "Joe", "Jane"],
...               'Height' : [1.82,1.65,2.15],
...               'Married': [False,False,True]}
>>> tbl = Tabel(data)
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 John   |     1.82 |         0
 Joe    |     1.65 |         0
 Jane   |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
```

From numpy arrays:

```
>>> Tabel([np.array(["John", "Joe", "Jane"]),
...                         np.array([1.82,1.65,2.15]),
...                         np.array([False,False,True])])
 0    |    1 |   2
------+------+-----
 John | 1.82 |   0
 Joe  | 1.65 |   0
 Jane | 2.15 |   1
3 rows ['<U4', '<f8', '|b1']
```

Or initialize an empty Tabel:

```
>>> tbl = Tabel()
>>> tbl
<BLANKLINE>
0 rows []
>>> len(tbl), tbl.shape
(0, (0, 0))
```

12.2 Slicing

API documentation: *tabel.Tabel.__getitem__*.

See the API reference for a detailed description: *Tabel.__getitem__*.

Some common examples:

```
>>> tbl = Tabel([ ["John", "Joe", "Jane"],
...                [1.82,1.65,2.15],
...                [False,False,True]], columns = ["Name", "Height", "Married"])
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 John   |     1.82 |         0
 Joe    |     1.65 |         0
 Jane   |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
```

Slicing works on both rows and columns:

```
>>> tbl[:,1:3]
   Height |   Married
----------+-----------
     1.82 |         0
     1.65 |         0
     2.15 |         1
3 rows ['<f8', '|b1']
```

Indexing:

```
>>> tbl[[1,2],[0,2]]
 Name   |   Married
--------+-----------
 Joe    |         0
 Jane   |         1
2 rows ['<U4', '|b1']
```

Using named columns:

```
>>> tbl[:,['Name','Married']]
 Name    |   Married
---------+-----------
 John    |          0
 Joe     |          0
 Jane    |          1
3 rows ['<U4', '|b1']
```

the ":" can be left out, if you're addressing columns by their names:

```
>>> tbl[:, ['Name','Married']]
 Name    |   Married
---------+-----------
 John    |          0
 Joe     |          0
 Jane    |          1
3 rows ['<U4', '|b1']
```

Indexing using boolean array's:

```
>>> index = ~tbl[:,'Married']
>>> tbl[index, :]
 Name    |   Height |   Married
---------+----------+-----------
 John    |     1.82 |          0
 Joe     |     1.65 |          0
2 rows ['<U4', '<f8', '|b1']
```

(The ":" can be omitted for columns as well)

If a single index is given for the row or colum position, the returned datatype is a row (tuple) or column (array) instead of Tabel:

```
>>> tbl['Married']
array([False, False,  True])
>>> tbl[1:2, 'Married']
array([False])
```

In all other cases the returned datatype is Tabel, including lists of length one:

```
>>> tbl[:, ['Married']]
   Married
-----------
          0
          0
          1
3 rows ['|b1']
```

Equally so for rows:

```
>>> tbl[2]
('Jane', 2.15, True)
>>> tbl[2,1:3]
(2.15, True)
```

Finally, single elements are obtained by individually addressing them:

```
>>> tbl[0,"Name"]
'John'
```

12.3 Setting

API documentation: *tabel.Tabel.__setitem__*.

There is an detailed description in the API documentation: *tabel.Tabel.__setitem__*. Generally, one just provides the datatype and shape that would have come from the equivalent get call:

Set a single element:

```
>>> tbl[0,"Name"] = "Jos"
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 Jos    |     1.82 |         0
 Joe    |     1.65 |         0
 Jane   |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
```

Set (part of) a column:

```
>>> tbl[0:2,1] = np.array([2,3])
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 Jos    |     2    |         0
 Joe    |     3    |         0
 Jane   |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
```

Set (part of) a single row:

```
>>> tbl[0] = ["John", 1.333, 0]
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 John   |    1.333 |         0
 Joe    |    3     |         0
 Jane   |    2.15  |         1
3 rows ['<U4', '<f8', '|b1']
```

12.4 Referencing

Slices are references, Slices return new table objects, but their data always refers back to the original one as long as that remains in existence. Exceptions are:

- New initialization of Tabel objects copy the data
- Boolean or integer indexing of *rows* returns a Tabel object with copied data

To show, take a slice from *tbl*, modify its first columns, check out the original *tbl*:

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.15],
↪'Married': [False,False,True]})
>>> tbl_b = tbl[:,[1,2]]
>>> tbl_b[:,0] = [1.3,1.4,1.5]
>>> tbl
 Name   |   Height |   Married
--------+----------+----------
 John   |      1.3 |         0
 Joe    |      1.4 |         0
 Jane   |      1.5 |         1
3 rows ['<U4', '<f8', '|b1']
```

Column arrays are references too, with the same exceptions as slices. Therefore the standard numpy arithmetic can be used:

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.15],
↪'Married': [False,False,True]})
>>> tbl["Height"] *= 2
>>> tbl
 Name   |   Height |   Married
--------+----------+----------
 John   |     3.64 |         0
 Joe    |     3.3  |         0
 Jane   |     4.3  |         1
3 rows ['<U4', '<f8', '|b1']
```

12.5 Appending

12.5.1 Appending Tabel

API documentation: *tabel.Tabel.append*.

Tabels can be appended with their append method:

```
>>> tbl = Tabel([ ["John", "Joe", "Jane"],
...                [1.82,1.65,2.15],
...                [False,False,True]], columns = ["Name", "Height", "Married"])
>>> tblb = Tabel([["Bas"],[2.01],[True]], columns=["Name", "Height", "Married"])
>>> tbl.append(tblb)
>>> tbl
 Name   |   Height |   Married
--------+----------+----------
 John   |     1.82 |         0
 Joe    |     1.65 |         0
 Jane   |     2.15 |         1
 Bas    |     2.01 |         1
4 rows ['<U4', '<f8', '|b1']
```

or using the "+=" syntax:

```
>>> tblb = Tabel([["Bas"],[2.01],[True]], columns=["Name", "Height", "Married"])
>>> tbl = Tabel([ ["John", "Joe", "Jane"],
...                [1.82,1.65,2.15],
...                [False,False,True]], columns = ["Name", "Height", "Married"])
```

(continues on next page)

(continued from previous page)

```
>>> tbl += tblb
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 John   |     1.82 |         0
 Joe    |     1.65 |         0
 Jane   |     2.15 |         1
 Bas    |     2.01 |         1
4 rows ['<U4', '<f8', '|b1']
```

12.5.2 Appending row

API documentation: *tabel.Tabel.row_append*.

You can also append a row (dict, list or tuple) at the end of the Tabel, for example:

```
>>> tbl.row_append({'Name':"Jack", 'Height':1.82, 'Married':1})
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 John   |     1.82 |         0
 Joe    |     1.65 |         0
 Jane   |     2.15 |         1
 Bas    |     2.01 |         1
 Jack   |     1.82 |         1
5 rows ['<U4', '<f8', '<i8']
```

12.5.3 Appending column

API documentation: *tabel.Tabel.__setitem__*.

To add a new column to the Tabel, just provide a new column name:

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.
↪15], 'Married': [False,False,True]})
>>> tbl["New"] = "Foo"
>>> tbl["Newer"] = list(range(3))
>>> tbl
 Name   |   Height |   Married | New    |   Newer
--------+----------+-----------+--------+---------
 John   |     1.82 |         0 | Foo    |       0
 Joe    |     1.65 |         0 | Foo    |       1
 Jane   |     2.15 |         1 | Foo    |       2
3 rows ['<U4', '<f8', '|b1', '<U3', '<i8']
```

Notes: When changing a column two syntaxes give approximately the same result, with, however, a noteable difference. Using a slice object ":" will change all elements of the column with the new element(s) provided. If just the colum name is provided, with no indication for row, than the whole column is replaced with the column provided.

```
>>> tbl = Tabel( [ ["John", "Joe", "Jane"],
...               [1.82,1.65,2.15],
...               [False,False,True] ],
...     columns = ["Name", "Height", "Married"])
```

(continues on next page)

(continued from previous page)

```
>>> tbl[:, "Name"] = [1, 2, 3]
>>> tbl
   Name |   Height |   Married
--------+----------+-----------
      1 |     1.82 |         0
      2 |     1.65 |         0
      3 |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
>>> tbl["Name"] = [1, 2, 3]
>>> tbl
   Name |   Height |   Married
--------+----------+-----------
      1 |     1.82 |         0
      2 |     1.65 |         0
      3 |     2.15 |         1
3 rows ['<i8', '<f8', '|b1']
```

Note how in the first case the type of the name column stays "<U8" while seccond case the type of the Name column changes to *<i8*.

12.6 Changing column names

API documentation: `tabel.Tabel.columns`.

Just manipulate the columns property directly:

```
>>> data = [ ["John", "Joe", "Jane"],
...                 [1.82,1.65,2.15],
...                 [False,False,True]]
>>> tbl = Tabel(data, columns=['Name','Height','Married'])
>>> tbl
 Name   |   Height |   Married
--------+----------+-----------
 John   |     1.82 |         0
 Joe    |     1.65 |         0
 Jane   |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
>>> tbl.columns = ["First Name", "BMI", "Overweght"]
>>> tbl
 First Name   |   BMI |   Overweght
--------------+-------+-------------
 John         |  1.82 |           0
 Joe          |  1.65 |           0
 Jane         |  2.15 |           1
3 rows ['<U4', '<f8', '|b1']
```

12.7 Transposing

API documentation: *tabel.T*.

Data from database connectors often comes in list of records, a convenience function is available to make the transpose:

```
>>> from tabel import T
>>> data = [['John', 1.82, False], ['Joe', 1.65, False], ['Jane', 2.15, True]]
>>> tbl = Tabel(T(data))
>>> tbl
 0    |    1 |   2
------+------+-----
 John | 1.82 |   0
 Joe  | 1.65 |   0
 Jane | 2.15 |   1
3 rows ['<U4', '<f8', '|b1']
```

12.8 Group By

API documentation: *tabel.Tabel.group_by*.

To group by unique elements in a column or unique combinations of elements in columns provide the column(s) as a list as the first argument. The second argument is a list of tuples for the aggregate functions and their columns:

```
>>> from tabel import first
>>> tbl = Tabel({'a':[10,20,30, 40]*3, 'b':["100","200"]*6, 'c':[100,200]*6})
>>> tbl
   a |   b |   c
-----+-----+-----
  10 | 100 | 100
  20 | 200 | 200
  30 | 100 | 100
  40 | 200 | 200
  10 | 100 | 100
  20 | 200 | 200
  30 | 100 | 100
  40 | 200 | 200
  10 | 100 | 100
  20 | 200 | 200
  30 | 100 | 100
  40 | 200 | 200
12 rows ['<i8', '<U3', '<i8']
>>> tbl.group_by(['b','a'], [(np.sum, 'a'), (first, 'c')])
   b |   a |   a_sum |   c_first
-----+-----+---------+-----------
 100 |  10 |      30 |       100
 200 |  20 |      60 |       200
 100 |  30 |      90 |       100
 200 |  40 |     120 |       200
4 rows ['<U3', '<i8', '<i8', '<i8']
```

first is a convenience function, for when aggregation should just take the first element.

12.9 Sorting

API documentation: *tabel.Tabel.sort*.

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.
↪15], 'Married': [False,False,True]})
>>> tbl.sort("Name")
>>> tbl
 Name    |   Height |   Married
---------+----------+-----------
 Jane    |     2.15 |         1
 Joe     |     1.65 |         0
 John    |     1.82 |         0
3 rows ['<U4', '<f8', '|b1']
```

Note that one can use indexing to reorder in any order:

```
>>> tbl[[2,1,0],[2,1,0]]
   Married |   Height | Name
-----------+----------+--------
         0 |     1.82 | John
         0 |     1.65 | Joe
         1 |     2.15 | Jane
3 rows ['|b1', '<f8', '<U4']
```

12.10 Joining

API documentation: *tabel.Tabel.join*.

To join another Tabel, provide the Tabe and the column or columns to use as a key for joining:

```
>>> tbl = Tabel({"a":list(range(4)), "b": ['a','b'] *2})
>>> tbl_b = Tabel({"a":list(range(4)), "c": ['d','e'] *2})
>>> tbl.join(tbl_b, "a")
   a | b_l   | c_r
-----+-------+-------
   0 | a     | d
   1 | b     | e
   2 | a     | d
   3 | b     | e
4 rows ['<i8', '<U1', '<U1']
```

12.11 Saving

API documentation: *tabel.Tabel.save*.

Data can be saved to disk in various formats:

```
>>> tbl = Tabel({'Name' : ["John", "Joe", "Jane"], 'Height' : [1.82,1.65,2.15],
↪'Married': [False,False,True]})
>>> tbl.save("test.csv", fmt="csv")
```

I recommend the numpy native 'npz' format:

```
>>> tbl.save("test.npz", fmt="npz")
```

12.12 Reading

API documentation: *tabel.read_tabel*.

Read from disk:

```
>>> from tabel import read_tabel
>>> t = read_tabel("test.csv", fmt="csv")
>>> t
 Name   |   Height | Married
--------+----------+-----------
 John   |     1.82 | False
 Joe    |     1.65 | False
 Jane   |     2.15 | True
3 rows ['<U4', '<f8', '<U5']
```

CHAPTER 13

API Reference

13.1 tabel module

13.1.1 transpose

tabel.**transpose**(*datastruct*)
Transpose rows and columns.

Convenience function. Usually DB connectors return data as a list of records, Tabel takes and internally stores data as a list of columns (column store). This function will transpose the list of records into a list of columns without a priori assuming anything about the datatype of each individual element.

Parameters **datastruct** (*list*) – list or tuple containing lists or tuples with the data for each row.

Returns transposed datastruct, list containing lists with the data for each column.

tabel.**T**(*datastruct*)
Convenience alias for *tabel.transpose*.

13.1.2 read_tabel

tabel.**read_tabel**(*filename*, *fmt=u'auto'*)
Read data from disk

Read data from disk and return a Tabel object.

Parameters

- **filename** (*str*) – filename sring, including path and extension.
- **fmt** (*str*) – format specifier, supports: 'csv', 'npz', 'gz'.

Returns Tabel object containing the data.

13.1.3 first

tabel.**first**(*array*)

Get the first element when doing a *tabel.Tabel.group_by*.

Arguments :

array (numpy ndarray) : numpy 1D array containing the subset of elements

Returns : The first element of ar

Examples : See Tabel.group_by for examples

13.2 class Tabel

class tabel.**Tabel**(*datastruct=None*, *columns=None*, *copy=True*)

Tabel datastructure

Data table with rows and columns, rows are numbered columns are named. Each column has its own datatype. Data is stored by columns (column store), fixed datatype per column, varyiable datatypes from column to column.

Parameters

- **datastruct** (*object*) – list, tuple, ndarray or dict of lists, tuples, ndarrays or elements; or a *pandas.DataFrame*. List of columns of data. See *tabel.T* for a convenience function to transpose a list of records.
- **columns** (*list of strings*) – Column names, ignored when keys are part of the datastruct (dict and *pandas.DataFrame*). Automatic names are generated, if omitted, as strings of column number.
- **copy** (*boolean*) – Wether to make a copy of the data or to reference to the current memory location (when possible), default: True

---

Notes

1. It is possible to create an empty Tabel instance and later add data using the *tabel.Tabel.append* and/or *tabel.Tabel.__setitem__* methods.
2. It is possibe to add or manipulate data directly through the instance attributes `tabel.Tabel.columns` and `tabel.Tabel.data`. One could use the *tabel.Tabel.valid* method to check wether the manipulated structure is still valid.
4. If one or more (but not all) of the columns contain a single element this element is repeated to match the length of the other columns.

Examples

To initialize a Tabel, call the constructor with the data in column lists:

```
>>> from tabel import Tabel
>>> Tabel( [ ["John", "Joe", "Jane"],
...          [1.82, 1.65, 2.15],
...          [False, False, True] ],
```

(continues on next page)

(continued from previous page)

```
...       columns = ["Name", "Height", "Married"])
 Name    |   Height |   Married
---------+----------+-----------
 John    |     1.82 |         0
 Joe     |     1.65 |         0
 Jane    |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
```

13.2.1 getter

Tabel.**__getitem__**(*key*)

Indexing and slicing parts of a Tabel.

Slicing and indexing mostly follows Numpy array and Python list conventions.

Arguments:

key (r, c): r can be a single integer, a boolean array, an integer itereable or a slice object. c can be a single integer or string, a boolean array, an integer or string itereable or a slice object.

key (int, string) : When only a single int or string is supplied it is considered to point to a whole single column or a whole single row (in that order).

Returns:

Depending on key, four different types can be returned.

element (): If both the row place and the column place are a single integer (or string for the column place), adressing a single element in the Tabel, wich could be of any datatype supported by Numpy.ndarray.

column (ndarray): If the column place is a single string or integer, adressing a single column and the row place is either abscent or not an integer.

row (tuple) : If the row place is a single integer, adressing a single row and the coumn place is either abscent or not a single integer/string.

Tabel (Tabel) : If a tuple key (r, c) is provided with anything other than an integer for the row place and anything other than a single integer/string type for the column place.

Notes

Returned Tabel objects from slicing are referenced to the original Tabel object unless row indexing was with a boolean list/array or the returned type was not a Tabel or np.ndarray object. Changes made to the slice will be reflected in the original Tabel. Appending or joining Tabels or adding/renaming columns will never be reflected in the original Tabel object. Use the *py:copy* function to make a full copy of the object.

Raises `KeyError` – When a key is referencing an invallid or not existing part of the data.

Examples

```
>>> tbl[:, 1:3]
   Height |   Married
----------+-----------
     1.82 |          0
     1.65 |          0
     2.15 |          1
3 rows ['<f8', '|b1']
```

```
>>> tbl[0, 0]
'John'
>>> tbl["Name"]
array(['John', 'Joe', 'Jane'], dtype='<U4')
>>> tbl[0]
('John', 1.82, False)
```

13.2.2 setter

Tabel.**__setitem__**(*key*, *value*)

Setting a slice of a Tabel

Setting, like getting, slices mostly follows numpy conventions. Specifically the rules for the key are the same as for *tabel.Tabel.__getitem__* with the same relation between key and expected type for the value. In adition this method can also be used to add new columns.

Parameters

- **key** (*int, string*) – r can be a single integer, a boolean array, an integer itereable or a slice object.

  c can be a single integer or string, a boolean array, an integer or string itereable or a slice object.

  To adress a single element in the Tabel object the key should be a tuple of (r, c) with r a single integer adressing the row and c a single integer or string addressing the column of the element to be changed.

- **key** – When only a single int or string is supplied it is considered to point to a whole single column or a whole single row (in that order).

- **value** (*object*) – The type the value needs to have depends on the key provided.

  element: A single element of the same type, or a type convertable to the same, as the column targeted as a destination. See *tabel.Tabel.dtype* to get the type of the columns.

  column : An array or list of elements, each element of of the same type, or a type convertable to the same, as the column targeted as a destination. If a new column is targeted a single element could be provided, in which case it will be replicated along all rows.

  row : A tuple of elements, each of the same type or a type convertable to the same, as the column targeted as a destination. Length of the tuple should match the number of columns addressed.

  Tabel : Not currently implemented.

Returns nothing, change in-place.

Notes

When changing a column two syntaxes give approximately the same result, with, however, a noteable difference. Using a slice object ":" will change all elements of the column with the new element(s) provided. If just the colum name is provided, with no indication for row, than the whole column is replaced with the column provided.

```
>>> tbl = Tabel( [ ["John", "Joe", "Jane"], [1.82, 1.65, 2.15],
...               [False, False, True] ], columns = ["Name", "Height", "Married"])
>>> tbl[:, "Name"] = [1, 2, 3]
>>> tbl
   Name |   Height |   Married
--------+----------+----------
      1 |     1.82 |         0
      2 |     1.65 |         0
      3 |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
>>> tbl["Name"] = [1, 2, 3]
>>> tbl
   Name |   Height |   Married
--------+----------+----------
      1 |     1.82 |         0
      2 |     1.65 |         0
      3 |     2.15 |         1
3 rows ['<i8', '<f8', '|b1']
```

Note how in the first case the type of the name column stays "<U8" while seccond case the type of the Name column changes to "<i8".

13.2.3 repr

Tabel.**__repr__**()

Pretty print using tabulate.

Examples

```
>>> tbl = Tabel( [ ["John", "Joe", "Jane"], [1.82, 1.65, 2.15],
...           [False, False, True] ], columns = ["Name", "Height", "Married"])
>>> tbl
 Name    |   Height |   Married
--------+----------+----------
 John    |     1.82 |         0
 Joe     |     1.65 |         0
 Jane    |     2.15 |         1
3 rows ['<U4', '<f8', '|b1']
```

13.2.4 append

Tabel.**append**(*tbl*)

Append new Tabel to the current Tabel.

Append a Tabel or pandas.DataFrame to the end of this Tabel. Each column is appended to each column of the instance invoking the method.

Parameters **tbl** (Tabel) – Tabel with the same columns as the current Tabel, order of columns does not need to match. Columns do not need to match if the current Tabel has zero length. Besides Tabel onjects pandas.DataFrame objects are also allowed.

Returns Nothing, change in-place.

13.2.5 row_append

Tabel.**row_append**(*row*)

Append a row reccord at the end of the Tabel.

Appending a single row at the end of the Tabel.

Parameters **row** (*dict, list, tuple*) – The row to be appended to the Tabel. If a dict is provided the keys should match the column names of the Tabel. If a list or tuple is provided the length and order should match the columns of the Tabel. columns do not need to match if the current Tabel has zero length.

Returns Nothing. Change in-place.

13.2.6 join

Tabel.**join**(*tbl_r*, *key*, *key_r=None*, *jointype=u'inner'*, *suffixes=(u'_l'*, *u'_r')*)

dbase join tables with ind column(s) as the keys.

Performs a database style joins on the two tables, the current instance and the provided tabel 'tbl_r' on the columns listed in 'key'.

Parameters

- **tbl_r** (Tabel) – The right tabel to be joined.
- **key** (*string or list*) – Name of the column(s) to be used as the key(s).
- **key_r** (*list*) – A list of columnnames of the right tabel matching the left tabel. Defaults to the list provided in *ind*.
- **jointype** (*str*) – One of: *inner*, *left*, *right*, *outer*. If *inner*, returns the elements common to both tabels. If *outer*, returns the common elements as well as the elements of the left tabel not in the right tabel and the elements of the right tabel not in the left tabel. If *left*, returns the common elements and the elements of the left tabel not in the right tabel. If *right*, returns the common elements and the elements of the right tabel not in the left tabel.
- **suffixes** (*tuple*) – Strings to be added to the left and right tabel column names.

Returns The joined tabel

Notes

The order and suffixes of the returned Tabel depend on the jointype. For all types, all but the key columns are suffixed with the left and the right suffix respectively. The left Tabel columns come first followed by the right Tabel columns, with the key column placed first of its Tabel columns. For *inner* and *left* jointypes the right key column is left out. for *right* jointype the left key column is left out. For the *outer* jointype both keys are present and suffixed.

Examples

Join a Tabel into the current Tabel matching on column 'a':

```
>>> tbl = Tabel({"a":list(range(4)), "b": ['a', 'b'] *2})
>>> tbl_b = Tabel({"a":list(range(4)), "c": ['d', 'e'] *2})
>>> tbl.join(tbl_b, "a")
   a | b_l   | c_r
-----+-------+-------
   0 | a     | d
   1 | b     | e
   2 | a     | d
   3 | b     | e
4 rows ['<i8', '<U1', '<U1']
```

13.2.7 group_by

Tabel.**group_by**(*key*, *aggregate_fie_col=None*)

Groups and aggregates Tabel.

Parameters

- **key** (*str or list*) – name or list of names of the columns to be grouped by.
- **aggregate_fie_col** (*list*) – list of tuples (*function*, *column*) where *function* is the function to be applied to aggregate and *column* is the string name of the column. *function* should take an 1D array as an input and the returned value is treated as a single element. Only the grouped columns of *key* are returned if ommited.

Returns Tabel object with requested columns

Examples

grouping by 'a' and then by 'b', agregating with taking the sum of 'a' elements and taking the first 'c' element of each group:

```
>>> tbl = Tabel({'a':[10, 20, 30, 40]*3, 'b':["100", "200"]*6, 'c':[100, 200]*6})
>>> from tabel import first
>>> tbl.group_by(['b', 'a'], [ (np.sum, 'a'), (first, 'c')])
   b |   a |   a_sum |   c_first
-----+-----+---------+-----------
 100 |  10 |      30 |       100
 200 |  20 |      60 |       200
 100 |  30 |      90 |       100
 200 |  40 |     120 |       200
4 rows ['<U3', '<i8', '<i8', '<i8']
```

13.2.8 sort

Tabel.**sort**(*columns*)

Sort the Tabel.

Sorting in-place the Tabel according to columns provided. Rows always stay together, just the order of rows is affectd.

Parameters **columns** (*string or list*) – column name or column names to be sorted, listed in-order.

Returns Nothing. Sorting in-place.

Examples

```
>>> tbl = Tabel({'a':['b', 'g', 'd'], 'b':list(range(3))})
>>> tbl.sort('a')
>>> tbl
 a   |   b
-----+-----
 b   |   0
 d   |   2
 g   |   1
3 rows ['<U1', '<i8']
```

13.2.9 astype

Tabel.**astype**(*dtypes*)

Returns a type-converted tabel.

Converts the tabel according to the provided list of dtypes and returns a new Tabel instance.

Parameters **dtypes** (*list*) – list of valid numpy dtypes in the order of the columns. List should have same length as number of columns present (see *Tabel.shape*) See Tabel.dtype for the current types of the Tabel.

Returns Tabel object with the columns converted to the new dtype.

Examples:

13.2.10 save

Tabel.**save**(*filename*, *fmt=u'auto'*)

Save to file

Saves the Tabel data including a header with the column names to a file of the specified name in the current directory or the directory specified.

Parameters

- **filename** (*str*) – filename, should include path
- **fmt** (*str*) – formatting, valid values are: 'auto', 'csv', 'npz', 'gz'

  auto : Determine the filetype from the fiel extension.

  csv : Write to csv file using pythons *csv* module.

  gz : Write to csv using pythons *csv* module and zip using standard *gzip* module.

  npz : Write to compressed *numpy* native binary format.

Returns Nothing.

13.2.11 properties

13.2.11.1 dict

Tabel.**dict**
Dump all data as a dict of columns.

Keywords are the column names and values are the column Numpy.ndarrays. Usefull when transferring to a pandas DataFrame.

13.2.11.2 shape

Tabel.**shape**
Tabel shape.

Returns tuple (r, c) with r the number of rows and c the number of columns.

13.2.11.3 len

Tabel.**__len__ = <unbound method Tabel.__len__>**

13.2.11.4 dtype

Tabel.**dtype**
List of dtypes of the data columns.

13.2.11.5 valid

Tabel.**valid**
Check wether the current datastructure is legit.

Returns (bool) True if the Tabel internal structure is valid.

Notes

This is currently checking for the length of the columns to be the same and the number of the columns to be the same as the number of column names.

13.2.12 class attributes

Tabel.**repr_layout = u'presto'**
The layout used with *tabulate* in the *__repr__()* method.

Type string

Tabel.**max_repr_rows = 20**
Maximum number of rows to show when *__repr__()* is invoked.

Type int

Tabel.**join_fill_value = {u'float': nan, u'integer': 999999, u'string': u''}**
Fill vallues to be used when doing outer joins

Type dict

CHAPTER 14

license

MIT License

Copyright (c) [2018] [Bastiaan Bergman]

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

D

F

G

J

M

R

S

T

V