tap Documentation

Release 1.0

Tobias Schmidt, Marco Biasini, Gabriel Studer

June 20, 2013

CONTENTS

This module defines the table class that provides convenient functionality to work with tabular data. It features functions to calculate statistical moments, e.g. mean, standard deviations as well as functionality to plot the data using matplotlib.

Contents:

CHAPTER

ONE

HANDLING OF TABULAR DATA

This document explains the basics of working with tabular data, and shows how to add, remove data from a table.

1.1 Basic Usage

Populate table with data:

```
from tap import *

# create table with two columns, x and y both of float type
tab=Tab(['x', 'y'], 'ff')
for x in range(1000):
  tab.add_row([x, x**2])

# create a plot
plt=tab.plot('x', 'y', save='x-vs-y.png')
```

Iterating over table items:

```
# load table from file
tab=load(...)

# iterate over all rows
for row in tab.rows:
  # print complete row
  print row

for f in tab.foo:
  print f
# iterate over all rows of selected columns
for foo, bar in tab.zip('foo','bar'):
  print foo, bar
```

1.2 Creating Table Objects

Tables can either be initialized with information from memory, or populated with data from `load`, e.g. CSV files or a pickled dump of a previously constructed table. An empty table can be easily constructed as follows

```
tab=Tab()
```

If you want to add columns directly when creating the table, column names and *column types* can be specified as follows

```
tab=Tab(['nameX','nameY','nameZ'], 'sfb')
```

this will create three columns called nameX, nameY and nameZ of type string, float and bool, respectively. When the second argument is omitted, the columns will all have a string type. There will be no data in the table and thus, the table will not contain any rows.

If you want to add data to the table in addition, use the following:

```
tab=Tab(['nameX','nameY','nameZ'],
          'sfb',
          nameX=['a','b','c'],
          nameY=[0.1, 1.2, 3.414],
          nameZ=[True, False, False])
```

If values for one column are omitted, they will be filled with NA, but if values are specified, all values must be specified (i.e. same number of values per column).

```
tab = Tab(['name, age'], 'string,float')
```

1.3 Column Types

Table columns have a specific type, e.g. string, float etc. Each cell in a column must either be of that type or set to not available (None). As a result, a float column can't contain string values. The following column types exist:

long name	abbreviation
string	s
float	f
int	i
bool	b

When adding new data to the table, values are automatically coerced (forced) to the column type. When coercing fails, a `ValueError` is thrown.

1.3.1 Specifying Column Types

The column types can be specified when initialing a new table. For convenience, several different formats are supported, which allow to specify the column types as strings, or list using long, or abbreviated forms. The following 5 examples initialise an empty table with a string, float, int and bool column each.

```
# abbreviated, compact form
tab = Tab(['x', 'y', 'z', 'u'], 'sfib')

# abbreviated, separated by coma
tab = Tab(['x', 'y', 'z', 'u'], 's, f, i, b')

# extended separated by coma
tab = Tab(['x', 'y', 'z', 'u'], 'string, float, int, bool')


# list abbreviated
tab = Tab(['x', 'y', 'z', 'u'], ['s', 'f', 'i', 'b'])
```

```
# list extended
tab = Tab(['x', 'y', 'z', 'u'], ['string', 'float', 'int', 'bool'])
```

1.3.2 Guessing Column Types

For the lazy, the table supports guessing the column type from data when initialising a new table. The detection of column types tries to convert each value to a particular type, e.g. int. When the type conversion is not successful for any value, the column type is set to string. As a special case, when the data arrays are empty, the array types are set to string.

```
# initialises a table with an bool and int column
t = Tab(['x','y'], x='True False False'.split(), y='1 NA 3'.split())
print t.col_types # bool int
```

1.4 Adding and Removing Data from a Table

The following methods allow to add and remove data in a row and column-wise manner.

`Tab.`**`add_row`**(*data*, *overwrite=None*)

Add a row to the table.

data may either be a dictionary or a list-like object:

- If *data* is a dictionary the keys in the dictionary must match the column names. Columns not found in the dict will be initialized to None. If the dict contains list-like objects, multiple rows will be added, if the number of items in all list-like objects is the same, otherwise a `ValueError` is raised.
- If *data* is a list-like object, the row is initialized from the values in *data*. The number of items in *data* must match the number of columns in the table. A `ValuerError` is raised otherwise. The values are added in the order specified in the list, thus, the order of the data must match the columns.

If *overwrite* is not None and set to an existing column name, the specified column in the table is searched for the first occurrence of a value matching the value of the column with the same name in the dictionary. If a matching value is found, the row is overwritten with the dictionary. If no matching row is found, a new row is appended to the table.

Parameters

- **data** (`dict` or *list-like* object) – data to add
- **overwrite** (`str`) – column name to overwrite existing row if value in column *overwrite* matches

Raises `ValueError` if *list-like* object is used and number of items does *not* match number of columns in table.

Raises `ValueError` if *dict* is used and multiple rows are added but the number of data items is different for different columns.

Example: add multiple data rows to a subset of columns using a dictionary

```
# create table with three float columns
tab = Tab(['x','y','z'], 'fff')

# add rows from dict
data = {'x': [1.2, 1.6], 'z': [1.6, 5.3]}
tab.add_row(data)
```

```
print tab

'''
will produce the table

====  ====  ====
x     y     z
====  ====  ====
1.20  NA    1.60
1.60  NA    5.30
====  ====  ====
'''

# overwrite the row with x=1.2 and add row with x=1.9
data = {'x': [1.2, 1.9], 'z': [7.9, 3.5]}
tab.add_row(data, overwrite='x')
print tab

'''
will produce the table

====  ====  ====
x     y     z
====  ====  ====
1.20  NA    7.90
1.60  NA    5.30
1.90  NA    3.50
====  ====  ====
'''
```

Tab.**add_col**(*col_name*, *col_type*, *data=None*)

Add a column to the right of the table.

Parameters

- **col_name** (str) – name of new column
- **col_type** (str) – type of new column (long versions: *int*, *float*, *bool*, *string* or short versions: *i*, *f*, *b*, *s*)
- **data** (*scalar or iterable*) – data to add to new column.

Example:

```
tab=Tab(['x'], 'f', x=range(5))
tab.add_col('even', 'bool', itertools.cycle([True, False]))
print tab

'''
will produce the table

====  ====
x     even
====  ====
  0   True
  1   False
  2   True
  3   False
  4   True
====  ====
```

```
'''
```

If data is a constant instead of an iterable object, it's value will be written into each row:

```
tab=Tab(['x'], 'f', x=range(5))
tab.add_col('num', 'i', 1)
print tab

'''
will produce the table

====  ====
x     num
====  ====
  0    1
  1    1
  2    1
  3    1
  4    1
====  ====
'''
```

As a special case, if there are no previous rows, and data is not None, rows are added for every item in data.

`Tab.`**`remove_col`**(*col*)
Remove column with the given name from the table

Parameters **col** (`str`) – name of column to remove

`Tab.`**`rename_col`**(*old_name*, *new_name*)
Rename column *old_name* to *new_name*.

Parameters

- **old_name** – Name of the old column
- **new_name** – Name of the new column

Raises `ValueError` when *old_name* is not a valid column

1.5 Combining Tables

`Tab.`**`extend`**(*tab*, *overwrite=None*)
Append each row of *tab* to the current table. The data is appended based on the column names, thus the order of the table columns is *not* relevant, only the header names.

If there is a column in *tab* that is not present in the current table, it is added to the current table and filled with *None* for all the rows present in the current table.

If the type of any column in *tab* is not the same as in the current table a *TypeError* is raised.

If *overwrite* is not None and set to an existing column name, the specified column in the table is searched for the first occurrence of a value matching the value of the column with the same name in the dictionary. If a matching value is found, the row is overwritten with the dictionary. If no matching row is found, a new row is appended to the table.

`tap.`**`merge`**(*table1*, *table2*, *by*, *only_matching=False*)
Returns a new table containing the data from both tables. The rows are combined based on the common values in the column(s) by. The option 'by' can be a list of column names. When this is the case, merging is based on multiple columns. For example, the two tables below

x	y
1	10
2	15
3	20

x	u
1	100
3	200
4	400

when merged by column x, it produces the following output:

x	y	u
1	10	100
2	15	None
3	20	200
4	None	400

1.6 Sorting/Querying data

`Tab.`**`to_string`**(*float_format='%.3f'*, *int_format='%d'*, *rows=None*)
Convert the table into a string representation.

The output format can be modified for int and float type columns by specifying a formatting string for the parameters 'float_format' and 'int_format'.

The option 'rows' specify the range of rows to be printed. The parameter must be a type that supports indexing (e.g. a `list`) containing the start and end row *index*, e.g. [start_row_idx, end_row_idx].

Parameters

- **float_format** (`str`) – formatting string for float columns
- **int_format** (`str`) – formatting string for int columns
- **rows** (iterable containing `ints`) – iterable containing start and end row *index*

`Tab.`**`sort`**(*by*, *order='+'*)
Performs an in-place sort of the table, based on column *by*.

Parameters

- **by** (`str`) – column name by which to sort
- **order** (`str` (i.e. +, -)) – ascending (−) or descending (+) order

`Tab.`**`empty`**(*col_name=None*, *ignore_nan=True*)
Checks if a table is empty.

If no column name is specified, the whole table is checked for being empty, whereas if a column name is specified, only this column is checked.

By default, all NAN (or None) values are ignored, and thus, a table containing only NAN values is considered as empty. By specifying the option ignore_nan=False, NAN values are counted as 'normal' values.

`Tab.`**`get_unique`**(*col*, *ignore_nan=True*)
Extract a list of all unique values from one column

Parameters

- **col** (`str`) – column name

- **ignore_nan** (`bool`) – ignore all *None* values

`Tab.`**`has_col`**(*col*)
Checks if the column with a given name is present in the table.

1.7 Accessing data

The data in a table can be iterated row and column-wise.

`Tab.`**`zip`**(**args*)
Allows to conveniently iterate over a selection of columns, e.g.

```
tab=Tab.load('...')
for col1, col2 in tab.zip('col1', 'col2'):
  print col1, col2
```

is a shortcut for

```
tab=Tab.load('...')
for col1, col2 in zip(tab['col1'], tab['col2']):
  print col1, col2
```

`Tab.`**`zip_non_null`**(**args*)
Same as `zip()`, but only returns rows where none of the values is None.

`Tab.`**`filter`**(**args*, ***kwargs*)
Returns a filtered table only containing rows matching all the predicates in kwargs and args For example,

```
tab.filter(town='Basel')
```

will return all the rows where the value of the column "town" is equal to "Basel". Several predicates may be combined, i.e.

```
tab.filter(town='Basel', male=True)
```

will return the rows with "town" equal to "Basel" and "male" equal to true. args are unary callables returning true if the row should be included in the result and false if not.

`Tab.`**`search_col_names`**(*regex*)
Returns a list of column names matching the regex

Parameters **regex** (`str`) – regex pattern

Returns `list` of column names (`str`)

CHAPTER

TWO

STATISTICS AND DATA ANALYSIS

Tab.**min**(*col*)
Returns the minimal value in col. None values are ignored.

Parameters **col** (str) – column name

Tab.**max**(*col*)
Returns the maximum value in col. None values are ignored.

Parameters **col** (str) – column name

Tab.**max_row**(*col*)
Returns the row containing the cell with the maximal value in col. If several rows have the highest value, only the first one is returned. None values are ignored.

Parameters **col** (str) – column name

Returns row with maximal col value or None if the table is empty

Tab.**min_row**(*col*)
Returns the row containing the cell with the minimal value in col. If several rows have the lowest value, only the first one is returned. None values are ignored.

Parameters **col** (str) – column name

Returns row with minimal col value or None if the table is empty

Tab.**mean**(*col*)
Returns the mean of the given column. Cells with None are ignored. Returns None, if the column doesn't contain any elements. Col must be of numeric ('float', 'int') or boolean column type.

If column type is *bool*, the function returns the ratio of number of 'Trues' by total number of elements.

Parameters **col** (str) – column name

Raises TypeError if column type is string

Tab.**median**(*col*)
Returns the median of the given column. Cells with None are ignored. Returns None, if the column doesn't contain any elements. Col must be of numeric column type ('float', 'int') or boolean column type.

Parameters **col** (str) – column name

Raises TypeError if column type is string

Tab.**std_dev**(*col*)
Returns the standard deviation of the given column. Cells with None are ignored. Returns None, if the column doesn't contain any elements. Col must be of numeric column type ('float', 'int') or boolean column type.

Parameters **col** (str) – column name

Raises `TypeError` if column type is `string`

`Tab.`**`count`**(*col*, *ignore_nan=True*)
count the number of cells in column that are not equal to None.

Parameters

- **col** (`str`) – column name
- **ignore_nan** (`bool`) – ignore all *None* values

`Tab.`**`correl`**(*col1*, *col2*)
Calculate the Pearson correlation coefficient between *col1* and *col2*, only taking rows into account where both of the values are not equal to *None*. If there are not enough data points to calculate a correlation coefficient, *None* is returned.

Parameters

- **col1** (`str`) – column name for first column
- **col2** (`str`) – column name for second column

`Tab.`**`spearman_correl`**(*col1*, *col2*)
Calculate the Spearman correlation coefficient between col1 and col2, only taking rows into account where both of the values are not equal to None. If there are not enough data points to calculate a correlation coefficient, None is returned.

Warning The function depends on the following module: *scipy.stats.mstats*

Parameters

- **col1** (`str`) – column name for first column
- **col2** (`str`) – column name for second column

`Tab.`**`compute_roc`**(*score_col*, *class_col*, *score_dir='-'*, *class_dir='-'*, *class_cutoff=2.0*)
Computes the receiver operating characteristics (ROC) of column *score_col* classified according to *class_col*.

For this it is necessary, that the datapoints are classified into positive and negative points. This can be done in two ways:

- by using one 'bool' column (*class_col*) which contains True for positives and False for negatives
- by using a non-bool column (*class_col*), a cutoff value (*class_cutoff*) and the classification columns direction (*class_dir*). This will generate the classification on the fly
  - if `class_dir=='-'`: values in the classification column that are less than or equal to *class_cutoff* will be counted as positives
  - if `class_dir=='+'`: values in the classification column that are larger than or equal to *class_cutoff* will be counted as positives

During the calculation, the table will be sorted according to *score_dir*, where a '-' values means smallest values first and therefore, the smaller the value, the better.

If *class_col* does not contain any positives (i.e. value is True (if column is of type bool) or evaluated to True (if column is of type int or float (depending on *class_dir* and *class_cutoff*))) the ROC is not defined and the function will return *None*.

Warning If either the value of *class_col* or *score_col* is *None*, the data in this row is ignored.

`Tab.`**`compute_enrichment`**(*score_col*, *class_col*, *score_dir='-'*, *class_dir='-'*, *class_cutoff=2.0*)
Computes the enrichment of column *score_col* classified according to *class_col*.

For this it is necessary, that the datapoints are classified into positive and negative points. This can be done in two ways:

- by using one 'bool' type column (*class_col*) which contains *True* for positives and *False* for negatives
- by specifying a classification column (*class_col*), a cutoff value (*class_cutoff*) and the classification columns direction (*class_dir*). This will generate the classification on the fly
  - if `class_dir=='-'`: values in the classification column that are less than or equal to class_cutoff will be counted as positives
  - if `class_dir=='+'`: values in the classification column that are larger than or equal to class_cutoff will be counted as positives

During the calculation, the table will be sorted according to *score_dir*, where a '-' values means smallest values first and therefore, the smaller the value, the better.

Warning If either the value of *class_col* or *score_col* is *None*, the data in this row is ignored.

`Tab.`**`compute_mcc`**(*score_col*, *class_col*, *score_dir='-'*, *class_dir='-'*, *score_cutoff=2.0*, *class_cutoff=2.0*)

Compute Matthews correlation coefficient (MCC) for one column (*score_col*) with the points classified into true positives, false positives, true negatives and false negatives according to a specified classification column (*class_col*).

The datapoints in *score_col* and *class_col* are classified into positive and negative points. This can be done in two ways:

- by using 'bool' columns which contains True for positives and False for negatives
- by using 'float' or 'int' columns and specifying a cutoff value and the columns direction. This will generate the classification on the fly
  - if `class_dir`/`score_dir=='-'`: values in the classification column that are less than or equal to *class_cutoff*/*score_cutoff* will be counted as positives
  - if `class_dir`/`score_dir=='+'`: values in the classification column that are larger than or equal to *class_cutoff*/*score_cutoff* will be counted as positives

The two possibilities can be used together, i.e. 'bool' type for one column and 'float'/'int' type and cut-off/direction for the other column.

`Tab.`**`optimal_prefactors`**(*ref_col*, **args*, ***kwargs*)

This returns the optimal prefactor values (i.e. a, b, c, ...) for the following equation

$$a * u + b * v + c * w + ... = z \tag{2.1}$$

where u, v, w and z are vectors. In matrix notation

$$A * p = z \tag{2.2}$$

where A contains the data from the table (u,v,w,...), p are the prefactors to optimize (a,b,c,...) and z is the vector containing the result of equation (2.1).

The parameter ref_col equals to z in both equations, and *args are columns u, v and w (or A in (2.2)). All columns must be specified by their names.

Example:

```
tab.optimal_prefactors('colC', 'colA', 'colB')
```

The function returns a list of containing the prefactors a, b, c, ... in the correct order (i.e. same as columns were specified in *args).

Weighting: If the kwarg weights="columX" is specified, the equations are weighted by the values in that column. Each row is multiplied by the weight in that row, which leads to (2.3):

$$weight * a * u + weight * b * v + weight * c * w + ... = weight * z \tag{2.3}$$

Weights must be float or int and can have any value. A value of 0 ignores this equation, a value of 1 means the same as no weight. If all weights are the same for each row, the same result will be obtained as with no weights.

Example:

```
tab.optimal_prefactors('colC', 'colA', 'colB', weights='colD')
```

Tab.**stats**(*col*)

Tab.**row_mean**(*mean_col_name*, *cols*)

Adds a new column of type 'float' with a specified name (*mean_col_name*), containing the mean of all specified columns for each row.

Cols are specified by their names and must be of numeric column type ('float', 'int') or boolean column type. Cells with None are ignored. Adds None if the row doesn't contain any values.

Parameters

- **mean_col_name** (`str`) – name of new column containing mean values
- **cols** (`str` or `list` of strings) – name or list of names of columns to include in computation of mean

Raises `TypeError` if column type of columns in *col* is `string`

== Example ==

Staring with the following table:

x	y	u
1	10	100
2	15	None
3	20	400

the code here adds a column with the name 'mean' to yield the table below:

x	y	u	mean
1	10	100	50.5
2	15	None	2
3	20	400	201.5

Tab.**sum**(*col*)

Returns the sum of the given column. Cells with None are ignored. Returns 0.0, if the column doesn't contain any elements. Col must be of numeric column type ('float', 'int') or boolean column type.

Parameters **col** (`str`) – column name

Raises `TypeError` if column type is `string`

Tab.**paired_t_test**(*col_a*, *col_b*)

Two-sided test for the null-hypothesis that two related samples have the same average (expected values)

Parameters

- **col_a** – First column
- **col_b** – Second column

Returns P-value between 0 and 1 that the two columns have the same average. The smaller the value, the less related the two columns are.

CHAPTER

THREE

DATA VISUALISATION

`Tab.plot`(*x*, *y=None*, *z=None*, *style='.'*, *x_title=None*, *y_title=None*, *z_title=None*, *x_range=None*, *y_range=None*, *z_range=None*, *color=None*, *legend=None*, *num_z_levels=10*, *z_contour=True*, *z_interpol='nn'*, *diag_line=False*, *labels=None*, *max_num_labels=None*, *title=None*, *clear=True*, *save=False*, ***kwargs*)

Function to plot values from your table in 1, 2 or 3 dimensions using Matplotlib

Parameters

- **x** (`str`) – column name for first dimension
- **y** (`str`) – column name for second dimension
- **z** (`str`) – column name for third dimension
- **style** (`str`) – symbol style (e.g. ., -, *x*, *o*, +, *). For a complete list check (matplotlib docu).
- **x_title** (`str`) – title for first dimension, if not specified it is automatically derived from column name
- **y_title** (`str`) – title for second dimension, if not specified it is automatically derived from column name
- **z_title** (`str`) – title for third dimension, if not specified it is automatically derived from column name
- **x_range** (`list` of length two) – start and end value for first dimension (e.g. [start_x, end_x])
- **y_range** (`list` of length two) – start and end value for second dimension (e.g. [start_y, end_y])
- **z_range** (`list` of length two) – start and end value for third dimension (e.g. [start_z, end_z])
- **color** (`str`) – color for data (e.g. *b*, *g*, *r*). For a complete list check (matplotlib docu).
- **legend** (`str`) – legend label for data series
- **num_z_levels** (`int`) – number of levels for third dimension
- **diag_line** (`bool`) – draw diagonal line
- **labels** (`str`) – column name containing labels to put on x-axis for one dimensional plot
- **max_num_labels** (`int`) – limit maximum number of labels
- **title** (`str`) – plot title, if not specified it is automatically derived from plotted column names
- **clear** (`bool`) – clear old data from plot

- **save** (str) – filename for saving plot
- **z_contour** (bool) – draw contour lines
- **z_interpol** (str) – interpolation method for 3-dimensional plot (one of 'nn', 'linear')
- ****kwargs** – additional arguments passed to matplotlib

Returns the matplotlib.pyplot module

Examples: simple plotting functions

```
tab=Table(['a','b','c','d'],'iffi', a=range(5,0,-1),
                                    b=[x/2.0 for x in range(1,6)],
                                    c=[math.cos(x) for x in range(0,5)],
                                    d=range(3,8))

# one dimensional plot of column 'd' vs. index
plt=tab.Plot('d')
plt.show()

# two dimensional plot of 'a' vs. 'c'
plt=tab.Plot('a', y='c', style='o-')
plt.show()

# three dimensional plot of 'a' vs. 'c' with values 'b'
plt=tab.Plot('a', y='c', z='b')
# manually save plot to file
plt.savefig("plot.png")
```

Tab.**plot_histogram**(*col*, *x_range=None*, *num_bins=10*, *normed=False*, *histtype='stepfilled'*, *align='mid'*, *x_title=None*, *y_title=None*, *title=None*, *clear=True*, *save=False*, *color=None*, *y_range=None*)

Create a histogram of the data in col for the range *x_range*, split into *num_bins* bins and plot it using Matplotlib.

Parameters

- **col** (str) – column name with data
- **x_range** (list of length two) – start and end value for first dimension (e.g. [start_x, end_x])
- **y_range** (list of length two) – start and end value for second dimension (e.g. [start_y, end_y])
- **num_bins** (int) – number of bins in range
- **color** (str) – Color to be used for the histogram. If not set, color will be determined by matplotlib
- **normed** (bool) – normalize histogram
- **histtype** (str) – type of histogram (i.e. *bar*, *barstacked*, *step*, *stepfilled*). See (matplotlib docu).
- **align** (str) – style of histogram (*left*, *mid*, *right*). See (matplotlib docu).
- **x_title** (str) – title for first dimension, if not specified it is automatically derived from column name
- **y_title** (str) – title for second dimension, if not specified it is automatically derived from column name
- **title** (str) – plot title, if not specified it is automatically derived from plotted column names

- **clear** (bool) – clear old data from plot
- **save** (str) – filename for saving plot

Examples: simple plotting functions

```
tab=Table(['a'],'f', a=[math.cos(x*0.01) for x in range(100)])

# one dimensional plot of column 'd' vs. index
plt=tab.plot_histogram('a')
plt.show()
```

Tab.**plot_hexbin**(*x*, *y*, *title=None*, *x_title=None*, *y_title=None*, *x_range=None*, *y_range=None*, *binning='log'*, *colormap='jet'*, *show_scalebar=False*, *scalebar_label=None*, *clear=True*, *save=False*, *show=False*)

Create a heatplot of the data in col x vs the data in col y using matplotlib

Parameters

- **x** (str) – column name with x data
- **y** (str) – column name with y data
- **title** (str) – title of the plot, will be generated automatically if set to None
- **x_title** – label of x-axis, will be generated automatically if set to None
- **y_title** – label of y-axis, will be generated automatically if set to None
- **x_range** (list of length two) – start and end value for first dimension (e.g. [start_x, end_x])
- **y_range** (list of length two) – start and end value for second dimension (e.g. [start_y, end_y])
- **binning** – type of binning. If set to None, the value of a hexbin will correspond to the number of datapoints falling into it. If set to 'log', the value will be the log with base 10 of the above value (log(i+1)). If an integer is provided, the number of a hexbin is equal the number of datapoints falling into it divided by the integer. If a list of values is provided, these values will be the lower bounds of the bins.
- **colormap** – colormap, that will be used. Value can be every colormap defined in matplotlib or an own defined colormap. You can either pass a string with the name of the matplotlib colormap or a colormap object.
- **show_scalebar** (bool) – If set to True, a scalebar according to the chosen colormap is shown
- **scalebar_label** (str) – Label of the scalebar
- **clear** (bool) – clear old data from plot
- **save** (str) – filename for saving plot
- **show** (bool) – directly show plot

Tab.**plot_bar**(*cols*, *x_labels=None*, *x_labels_rotation='horizontal'*, *y_title=None*, *title=None*, *colors=None*, *yerr_cols=None*, *width=0.8*, *bottom=0*, *legend=True*, *save=False*)

Create a barplot of the data in cols. Every element of a column will be represented as a single bar. If there are several columns, each row will be grouped together.

Parameters

- **cols** – Column names with data. If cols is a string, every element of that column will be represented as a single bar. If cols is a list, every row resulting of these columns will be grouped together. Every value of the table still is represented by a single bar.
- **x_labels** (`list`) – Label for every row on x-axis.
- **x_labels_rotation** – Can either be 'horizontal', 'vertical' or a number that describes the rotation in degrees.
- **y_title** (`str`) – Y-axis description
- **colors** (`list`) – Colors of the different bars in each group. Must be a list of valid colornames in matplotlib. Length of color and cols must be consistent.
- **yerr_cols** – Columns containing the y-error information. Can either be a string if only one column is plotted or a list otherwise. Length of yerr_cols and cols must be consistent.
- **width** (`float`) – The available space for the groups on the x-axis is divided by the exact number of groups. The parameters width is the fraction of what is actually used. If it would be 1.0 the bars of the different groups would touch each other.
- **bottom** (`float`) – Bottom
- **legend** (`bool`) – Legend for color explanation, the corresponding column respectively.
- **save** (`str`) – If set, a png image with name $save in the current working directory will be saved.

Title Title

`Tab.`**`plot_enrichment`**(*score_col*, *class_col*, *score_dir='-'*, *class_dir='-'*, *class_cutoff=2.0*, *style='-'*, *title=None*, *x_title=None*, *y_title=None*, *clear=True*, *save=None*)

Plot an enrichment curve using matplotlib of column *score_col* classified according to *class_col*.

For more information about parameters of the enrichment, see `compute_enrichment()`, and for plotting see `Plot()`.

Warning The function depends on *matplotlib*

`Tab.`**`plot_roc`**(*score_col*, *class_col*, *score_dir='-'*, *class_dir='-'*, *class_cutoff=2.0*, *style='-'*, *title=None*, *x_title=None*, *y_title=None*, *clear=True*, *save=None*)

Plot an ROC curve using matplotlib.

For more information about parameters of the ROC, see `compute_roc()`, and for plotting see `Plot()`.

Warning The function depends on *matplotlib*

CHAPTER

FOUR

TAP - IMPORTING AND EXPORTING TABULAR DATA

The Tab class supports importing and exporting of tabular data from various sources. Import is achieved through the load() function, export through the table's Tab.save() method.

Example:

```
# load data from comma separated value file using ',' as the separator
tab = tap.load('data.csv', sep=',')
```

tap.**load**(*stream_or_filename*, *format='auto'*, *sep=','*)

Load table from an input stream or the file pointed to by filename.

By default, the file format is set to *auto*, which tries to guess the file format from the file extension. The following file extensions are recognized:

extension	recognized format
.csv	comma separated values
.pickle	pickled byte stream
<all others>	ost-specific format

For csv and pickle files with other file extensions, the format must be specified explicitly by setting *format* the appropriate format string.

The following file formats are understood:

- ost

  This is an ost-specific, but still human readable file format. The file (stream) must start with header line of the form

  col_name1[type1] <col_name2[type2]>...

  The types given in brackets must be one of the data types the Tab class understands. Each following line in the file then must contains exactly the same number of data items as listed in the header. The data items are automatically converted to the column format. Lines starting with a '#' and empty lines are ignored.

- pickle

  Deserializes the table from a pickled byte stream

- csv

  Reads the table from comma separated values stream. Since there is no explicit type information in the csv file, the column types are guessed, using the following simple rules:

  – if all values are either NA/NULL/NONE the type is set to string

–if all non-null values are convertible to float/int the type is set to float/int

–if all non-null values are true/false/yes/no, the value is set to bool

–for all other cases, the column type is set to string

Returns A new `Tab` instance

`Tab.`**`save`**(*stream_or_filename*, *format='ost'*, *sep=','*)

save the table to stream or file pointed to by filename. The following three file formats are supported (for more information on file formats, see `load()`):

ost	ost-specific format (human readable)
csv	comma separated values (human readable)
pickle	pickled byte stream (binary)
html	HTML table
context	ConTeXt table

Parameters

- **stream_or_filename** (`str` or `file`) – filename or stream for writing output
- **format** (`str`) – output format (i.e. *ost*, *csv*, *pickle*)

Raises `ValueError` if format is unknown

CHAPTER

FIVE

THE TAP API AT A GLANCE

Adding/Removing/Reordering data	
add_row()	add a row to the table
add_col()	add a column to the table
remove_col()	remove a column from the table
rename_col()	rename a column
extend()	append a table to the end of another table
merge()	merge two tables together
sort()	sort table by column
filter()	filter table by values
zip()	extract multiple columns at once
zip_non_null()	extract multiple columns at once, ignoring none
search_col_names()	search for matching column names
empty()	check whether table/column is empty
get_unique()	get unique values of a column
has_col()	check for existence of column
Input/Output	
save()	save a table to a file
load()	load a table from a file
to_string()	convert a table to a string for printing
Simple Math	
min()	compute the minimum of a column
max()	compute the maximum of a column
sum()	compute the sum of a column
mean()	compute the mean of a column
row_mean()	compute the mean for each row
median()	compute the median of a column
std_dev()	compute the standard deviation of a column
count()	compute the number of items in a column
More Sophisticated Math	
correl()	compute Pearson's correlation coefficient
spearman_correl()	compute Spearman's rank correlation coefficient
compute_mcc()	compute Matthew's correlation coefficient
compute_roc()	compute receiver operating characteristics (ROC)
compute_enrichment()	compute enrichment
optimal_prefactors()	compute optimal coefficients for linear combination of columns
stats()	get various statistics on a column
paired_t_test()	perform paired t-test on two columns

Continued on next page

Table 5.1 – continued from previous page

Plot	
`plot()`	Plot data in 1, 2 or 3 dimensions
`plot_histogram()`	Plot data as histogram
`plot_roc()`	Plot receiver operating characteristics (ROC)
`plot_enrichment()`	Plot enrichment
`plot_hexbin()`	Hexagonal density plot
`plot_bar()`	Bar plot

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

t