RecSQL Documentation

Release 0.7.12-dev

Oliver Beckstein

May 29, 2016

Contents

Release 0.7.12-dev

Date May 29, 2016

RecSQL is a hack that allows one to load table-like data records into an in-memory sqlite database for quick and dirty analysis via SQL. The *SQLarray* class has additional SQL functions such as `sqrt` or `histogram` defined. SQL tables can always be returned as numpy record arrays so that data can be easily handled in other packages such as numpy or plotted via matplotlib. Selections produce new *SQLarray* objects.

Most of the Documentation is generated from the python doc strings. See INSTALL for installation instructions.

The package and the documentation are still in flux and any feedback, bug reports, suggestions and contributions are very welcome. See the package README for contact details.

CHAPTER 1

License

This package is Copyright 2007–2016, Oliver Beckstein.

The **RecSQL** package is made available under the terms of the GNU General Public License v3 (or any higher version at your choice). See the file `LICENSE`.

CHAPTER 2

Contents

2.1 README

RecSQL's basic idea is to treat numpy record arrays like SQL tables. What it does, in fact, is to represent the arrays as real SQL tables (using SQLite) and provide convenience functions to return recarrays on demand.

This works ok for small tables but less so if you want to access gigabytes of data as recarrays. It's a hack.

Documentation can be found at http://recsql.readthedocs.org/

Source code is available from https://github.com/orbeckst/RecSQL under the GNU General Public License, version 3 (see also the file LICENSE in the distribution).

2.1.1 Example

```
>>> from recsql import SQLarray
>>> import numpy
>>> a = numpy.rec.fromrecords(numpy.arange(100).reshape(25,4), names='a,b,c,d')
>>> Q = SQLarray('my_name', a)
>>> print repr(Q.recarray)
rec.array([(0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11), (12, 13, 14, 15),
       (16, 17, 18, 19), (20, 21, 22, 23), (24, 25, 26, 27),
       (28, 29, 30, 31), (32, 33, 34, 35), (36, 37, 38, 39),
       (40, 41, 42, 43), (44, 45, 46, 47), (48, 49, 50, 51),
       (52, 53, 54, 55), (56, 57, 58, 59), (60, 61, 62, 63),
       (64, 65, 66, 67), (68, 69, 70, 71), (72, 73, 74, 75),
       (76, 77, 78, 79), (80, 81, 82, 83), (84, 85, 86, 87),
       (88, 89, 90, 91), (92, 93, 94, 95), (96, 97, 98, 99)],
      dtype=[('a', '<i4'), ('b', '<i4'), ('c', '<i4'), ('d', '<i4')])
>>> Q.SELECT('*', 'WHERE a < 10 AND b > 5')
rec.array([(8, 9, 10, 11)],
    dtype=[('a', '<i4'), ('b', '<i4'), ('c', '<i4'), ('d', '<i4')])
# creating new SQLarrays:
>>> R = Q.selection('a < 20 AND b > 5')
>>> print R
<recsql.sqlarray.SQLarray object at 0x...>
```

2.1.2 Availability

The latest version of the package is available on GitHub https://github.com/orbeckst/RecSQL

RecSQL is also listed on PyPi http://pypi.python.org/pypi/RecSQL and can thus be installed with

```
pip install RecSQL
```

See INSTALL for further installation instructions.

A git repository of the package is hosted at http://github.com/orbeckst/RecSQL .

2.1.3 Getting Involved

Please submit problems, questions and questions through the issue tracker. Pull requests are also very welcome.

2.2 INSTALL

You can install to the latest version directly from the internet (via the Python Package index, where RecSQl is listed as http://pypi.python.org/pypi/RecSQL) with

```
pip install RecSQL
```

You can also download the source manually from https://github.com/orbeckst/RecSQL/releases and unpack it. Install from the unpacked source with

```
cd RecSQL-0.7.11
python setup.py install
```

The latest sources can be obtained by cloning the RecSQL github repository

```
git clone git://github.com/orbeckst/RecSQL.git
```

and installing from the source as above.

Additional requirements are numpy (and pysqlite for older versions of Python). pip will automatically attempt to download appropriate versions if none are currently installed.

2.3 User documentation

2.3.1 RecSQL package

RecSQL is a simple module that provides a numpy.record array frontend to an underlying SQLite table.

The *`SQLarray`* object populates a SQL table from a numpy record array, a iterable that supplies table records, or a string that contains an especially simple reStructured text table. The SQL table is held in memory and functions are provided to run SQL queries and commands on the underlying database. Queries return record arrays if possible (although a flag can explicitly change this).

Query results are cached to improve performance. This can be disabled (which is recommened for large data sets).

The SQL table is named on initialization. Later one can refer to this table by the name or the magic name __*self*__ in SQL statements. Additional tables can be added to the same database (by using the connection keyword of the constructor)

The *`recsql.rest_table`* module uses the base functionality to parse a restructured text table from a string (such as a doc string) and returns a nicely structured table. This allows for use of parameters that are documented in the doc strings.

See also:

PyTables is a high-performance interface to table data. In most cases you will probably better off in the long run using PyTables than recSQL.

Important functions and classes

A *SQLarray* can be constructed by either reading data from a CSV file or reST table with the *SQLarray_fromfile()* function or constructed directly from a `numpy.recarray` via the *SQLarray* constructor.

`recsql.`**`SQLarray_fromfile`**(*filename*, ***kwargs*)
Create a *SQLarray* from *filename*.

Uses the filename suffix to detect the contents:

rst, txt restructure text (see *recsql.rest_table*

csv comma-separated (see *recsql.csv_table*)

Arguments

filename name of the file that contains the data with the appropriate file extension

kwargs

- additional arguments for *SQLarray*
- additional arguments *recsql.csv_table.Table2array* or *recsql.rest_table.Table2array* such as *mode* or *autoncovert*.

class `recsql.`**`SQLarray`**(*name=None*, *records=None*, *filename=None*, *columns=None*, *cachesize=5*, *connection=None*, *is_tmp=False*, ***kwargs*)
A SQL table that returns (mostly) rec arrays.

The *SQLarray* can be initialized from

1. an iterable of records (tuples), given in the *records* keyword argument, and the column names (provided in *columns*);
2. a string that contains a simple reStructured Text table (see *recsql.rest_table* for details);
3. a `numpy.recarray`.

Note: SQLite only understands standard Python types and hence has problems with many of the NumPy data types such as `numpy.int64`. When loading a recarray fails we try to convert all data types automatically to Python types (using *recsql.convert.irecarray_to_py()*). This might loose precision and/or even fail. It is also slow for larger arrays.

The class takes the following arguments:

`SQLarray`([*name*[, *records*[, *columns*[, *cachesize=5*, *connection=None*, *dbfile=":memory:"*]]]])

Arguments

name table name (can be referred to as '__self__' in SQL queries)

records numpy record array that describes the layout and initializes the table OR any iterable (and then columns must be set, too) OR a string that contains a single, *simple reStructured* *text table* (and the table name is set from the table name in the reST table.) If `None` then simply associate with existing table name.

filename Alternatively to *records*, read a reStructured table from *filename*.

columns sequence of column names (only used if records does not have attribute dtype.names) [`None`]

cachesize number of (query, result) pairs that are cached [5]

connection If not `None`, reuse this connection; this adds a new table to the same database, which allows more complicated queries with cross-joins. The table's connection is available as the attribute T.connection. [`None`]

dbfile Normally the db is held in memory (":memory:") but if a filename is provided then the underlying SQLite db is held on disk and can be accessed and restored (see `SQLarray.save()`). Only works with *connection* = `None` [":memory:"]

is_tmp `True`: create a tmp table; `False`: regular table in db [`False`]

Bugs

- `InterfaceError`: *Error binding parameter 0 - probably unsupported type*

  In this case the recarray contained types such as `numpy.int64` that are not understood by sqlite and which we were not able to convert to a Python type (using `recsql.convert.irecarray_to_py()`). Either convert the data manually (by setting the numpy dtypes yourself on the recarray, or better: feed a simple list of tuples ("records") to this class in *records*. Make sure that these tuples only contain standard Python types. Together with *records* you will also have to supply the names of the data columns in the keyword argument *columns*.

  If you are reading from a file then it might be simpler to use `recsql.sqlarray.SQLarray_fromfile()`.

SELECT (*fields*, **args*, ***kwargs*)

Execute a simple SQL `SELECT` statement and returns values as new numpy rec array.

The arguments *fields* and the additional optional arguments are simply concatenated with additional SQL statements according to the template:

```
SELECT <fields> FROM __self__ [args]
```

The simplest fields argument is `"*"`.

Example: Create a recarray in which students with average grade less than 3 are listed:

```
result = T.SELECT("surname, subject, year, avg(grade) AS avg_grade",
                  "WHERE avg_grade < 3", "GROUP BY surname,subject",
                  "ORDER BY avg_grade,surname")
```

The resulting SQL would be:

```
SELECT surname, subject, year, avg(grade) AS avg_grade FROM __self__
     WHERE avg_grade < 3
     GROUP BY surname,subject
     ORDER BY avg_grade,surname
```

Note how one can use aggregate functions such avg().

The string '*__self__*' is automatically replaced with the table name (`T.name`); this can be used for cartesian products such as

```
        LEFT JOIN __self__ WHERE ...
```

Note: See the documentation for `sql()` for more details on the available keyword arguments and the use of `?` parameter interpolation.

`close`()
Clean up (if no more connections to the db exist).

- For in-memory: Delete the underlying SQL table from the in-memory database.
- For on-disk: save and close connection

`connection_count`
Number of currently open connections to the database.

(Stored in table sqlarray_master.)

`has_table`(*name*)
Return `True` if the table *name* exists in the database.

`limits`(*variable*)
Return minimum and maximum of variable across all rows of data.

`merge`(*recarray*, *columns=None*)
Merge another recarray with the same columns into this table.

Arguments

recarray numpy record array that describes the layout and initializes the table

Returns n number of inserted rows

Raises Raises an exception if duplicate and incompatible data exist in the main table and the new one.

`merge_table`(*name*)
Merge an existing table in the database with the __self__ table.

Executes as `'INSERT INTO __self__ SELECT * FROM <name>'`. However, this method is probably used less often than the simpler `merge()`.

Arguments name name of the table in the database (must be compatible with __self__)

Returns n number of inserted rows

`recarray`
Return underlying SQL table as a read-only record array.

`save`()
Commit changes to file.

Only works if the SQLarray was created with they *dbfile* = `FILENAME` keyword. There is currently no way to save a in-memory db.

See also:

`aoft.DB.clone()`

`selection`(*SQL*, *parameters=None*, ***kwargs*)
Return a new SQLarray from a SELECT selection.

This method is useful to build complicated selections and essentially new tables from existing data. The result of the SQL query is stored as a new table in the database. By default, a unique name is created but this can be overridden with the *name* keyword.

Arguments

SQL SQL `SELECT` query string. A leading `SELECT * FROM __self__ WHERE` can be omitted (see examples below). The SQL is scrubbed and only data up to the first semicolon is used (note that this means that there cannot even be a semicolon in quotes; if this is a problem, file a bug report and it might be changed).

Keywords

name name of the table, `None` autogenerates a name unique to this query. *name* may not refer to the parent table itself. [`None`]

parameters tuple of values that are safely interpolated into subsequent `?` characters in the SQL string

force If `True` then an existing table of the same *name* is `DROP``ped first.  If ``False` and the table already exists then *SQL* is ignored and a *`SQLarray`* of the existing table *name* is returned. [`False`]

Returns a *`SQLarray`* referring to the table *name* in the database; it also inherits the `SQLarray.dbfile`

Examples:

```
s = SQLarray.selection('a > 3')
s = SQLarray.selection('a > ?', (3,))
s = SQLarray.selection('SELECT * FROM __self__ WHERE a > ? AND b < ?', (3, 10))
```

`sql`(*SQL*, *parameters=None*, *asrecarray=True*, *cache=True*)
Execute sql statement.

Arguments

SQL [string] Full SQL command; can contain the `?` place holder so that values supplied with the `parameters` keyword can be interpolated using the `pysqlite` interface.

parameters [tuple] Parameters for `?` interpolation.

asrecarray [boolean] `True`: return a `numpy.recarray` if possible; `False`: return records as a list of tuples. [`True`]

cache [boolean] Should the results be cached? Set to `False` for large queries to avoid memory issues. Queries with `?` place holders are never cached. [`True`]

Returns For *asrecarray* = `True` a `numpy.recarray` is returned; otherwise a simple list of tuples is returned.

Raises `TypeError` if the conversion to `recarray` fails for any reason.

Warning: There are **no sanity checks** applied to the SQL.

The last *cachesize* queries are cached (for *cache* = `True`) and are returned directly unless the table has been modified.

The string "__self__" in *SQL* is substituted with the table name. See the *`SELECT()`* method for more details.

sql_index (*index_name*, *column_names*, *unique=True*)
Add a named index on given columns to improve performance.

sql_select (*fields*, **args*, ***kwargs*)
Execute a simple SQL `SELECT` statement and returns values as new numpy rec array.

The arguments *fields* and the additional optional arguments are simply concatenated with additional SQL statements according to the template:

```
SELECT <fields> FROM __self__ [args]
```

The simplest fields argument is `"*"`.

Example: Create a recarray in which students with average grade less than 3 are listed:

```
result = T.SELECT("surname, subject, year, avg(grade) AS avg_grade",
                  "WHERE avg_grade < 3", "GROUP BY surname,subject",
                  "ORDER BY avg_grade,surname")
```

The resulting SQL would be:

```
SELECT surname, subject, year, avg(grade) AS avg_grade FROM __self__
     WHERE avg_grade < 3
     GROUP BY surname,subject
     ORDER BY avg_grade,surname
```

Note how one can use aggregate functions such avg().

The string '*__self__*' is automatically replaced with the table name (`T.name`); this can be used for cartesian products such as

```
LEFT JOIN __self__ WHERE ...
```

Note: See the documentation for `sql()` for more details on the available keyword arguments and the use of `?` parameter interpolation.

For querying the version of the package use

`recsql.`**get_version**()
Return current package version as a string.

`recsql.`**get_version_tuple**()
Return current package version as a (MAJOR,MINOR,PATCHLEVEL).

Example

```
>>> from recsql import SQLarray
>>> import numpy
>>> a = numpy.rec.fromrecords(numpy.arange(100).reshape(25,4), names='a,b,c,d')
>>> Q = SQLarray('my_name', a)
>>> print repr(Q.recarray)
rec.array([(0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11), (12, 13, 14, 15),
       (16, 17, 18, 19), (20, 21, 22, 23), (24, 25, 26, 27),
       (28, 29, 30, 31), (32, 33, 34, 35), (36, 37, 38, 39),
       (40, 41, 42, 43), (44, 45, 46, 47), (48, 49, 50, 51),
       (52, 53, 54, 55), (56, 57, 58, 59), (60, 61, 62, 63),
       (64, 65, 66, 67), (68, 69, 70, 71), (72, 73, 74, 75),
        (76, 77, 78, 79), (80, 81, 82, 83), (84, 85, 86, 87),
        (88, 89, 90, 91), (92, 93, 94, 95), (96, 97, 98, 99)],
       dtype=[('a', '<i4'), ('b', '<i4'), ('c', '<i4'), ('d', '<i4')])
>>> Q.SELECT('*', 'WHERE a < 10 AND b > 5')
rec.array([(8, 9, 10, 11)],
    dtype=[('a', '<i4'), ('b', '<i4'), ('c', '<i4'), ('d', '<i4')])
# creating new SQLarrays:
>>> R = Q.selection('a < 20 AND b > 5')
>>> print R
<recsql.sqlarray.SQLarray object at 0x...>
```

Additional SQL functions

Note that the SQL database that is used as the backend for *SQLarray* has a few additional functions defined in addition to the standard SQL available in sqlite. These can be used in `SELECT` statements and often avoid post-processing of record arrays in python. It is relatively straightforward to add new functions (see the source code and in particular the `recsql.sqlarray.SQLarray._init_sql_functions()` method; the functions themselves are defined in the module *recsql.sqlfunctions*).

Simple SQL functions

Simple functions transform a single input value into a single output value:

Expression	SQL equivalent
y = f(x)	SELECT f(x) AS y

Additional simple functions have been defined:

Simple SQL f()	description
sqr(x)	square x*x
sqrt(x)	square root `numpy.sqrt()`
pow(x,y)	power x**y
periodic(x)	wrap angle in degree between -180º and +180º
regexp(pattern,string)	string REGEXP pattern
match(pattern,string)	string MATCH pattern (anchored REGEXP)
fformat(format,x)	string formatting of a single value format % x

Aggregate SQL functions

Aggregate functions combine data from a query; they are typically used with a 'GROUP BY col' clause. They can be thought of as numpy ufuncs:

Expression	SQL equivalent
y = f(x1,x2,...xN)	SELECT f(x) AS y ... GROUP BY x

For completeness, the table also lists sqlite built-in aggregate functions:

Simple aggregate f()	description
avg(x)	mean [sqlite builtin]
std(x)	standard deviation (using N-1 variance)
stdN(x)	standard deviation (using N variance), sqrt(<(X - <X>)**2>)
median(x)	median of the data (see `numpy.median()`)
min(x)	minimum [sqlite builtin]
max(x)	maximum [sqlite builtin]

PyAggregate SQL functions

PyAggregate functions act on a list of data points in the same way as ordinary aggregate functions but they return python objects such as numpy arrays, or tuples of numpy arrays (eg bin edges and histogram). In order to make this work, specific types have to be declared when returning the results:

For instance, the histogram() function returns a python Object, the tuple (histogram, edges):

```
a.sql('SELECT histogram(x) AS "x [Object]" FROM __self__', asrecarray=False)
```

The return type ('Object') needs to be declared with the `'AS "x [Object]"'` syntax (note the quotes). (See more details in the sqlite documentation under adapters and converters.) The following table lists all *PyAggregate* functions that have been defined:

PyAggregate	type	signature; description
array	NumpyArray	array(x); a standard `numpy.array()`
histogram	Object	histogram(x,nbins,xmin,xmax); histogram x in nbins evenly spaced bins between xmin and xmax
distribution	Object	distribution(x,nbins,xmin,xmax); normalized histogram whose integral gives 1
meanhistogram	Object	meanhistogram(x,y,nbins,xmin,xmax); histogram data points y along x and average all y in each bin
stdhistogram	Object	stdhistogram(x,y,nbins,xmin,xmax); give the standard deviation (from N-1 variance) std(y) = sqrt(Var(y)) with Var(y) = <(y-<y>)^2>
medianhistogram	Object	medianhistogram((x,y,nbins,xmin,xmax); median(y)
minhistogram	Object	minhistogram((x,y,nbins,xmin,xmax); min(y)
maxhistogram	Object	maxhistogram((x,y,nbins,xmin,xmax); max(y)
zscorehistogram	Object	zscorehistogram((x,y,nbins,xmin,xmax); <abs(y-<y>)>/std(y)

Examples of using types in tables

The following show how to use the special types.

Declare types as 'NumpyArray':

```
a.sql("CREATE TABLE __self__(a NumpyArray)")
```

Then you can simply insert python objects (`type(my_array) == numpy.ndarray`):

```
a.sql("INSERT INTO __self__(a) values (?)", (my_array,))
```

When returning results of declared columns one does not have to do anything

```
(my_array,) = a.sql("SELECT a FROM __self__")
```

although one can also do

```
(my_array,) = q.sql('SELECT a AS "a [NumpyArray]" FROM __self__')
```

but when using a PyAggregate the type *must* be declared:

```
a.sql('SELECT histogram(x,10,0.0,1.5) as "hist [Object]" FROM __self__')
```

Other approaches to interfacing SQLite and NumPy

If RecSQL does not what you need it to do then look at these other projects.

See also:

esutil.sqlite_util (part of esutil) and hydroclimpy.io.sqlite

If you do not have to rely on SQL then also look at PyTables.

2.4 Notes for developers

2.4.1 `sqlarray` — Implementation of `SQLarray`

SQLarray is a thin wrapper around pysqlite SQL tables. The main features ares that `SELECT` queries can return `numpy.recarrays` and the *SQLarray.selection()* method returns a new *SQLarray* instance.

numpy arrays can be stored in sql fields which allows advanced table aggregate functions such as `histogram`.

A number of additional SQL functions are defined.

TODO

- Make object saveable (i.e. store the database on disk instead of memory or dump the memory db and provide a load() method
- Use hooks for the pickling protocol to make this transparent.

See also:

PyTables is a high-performance interface to table data.

Module content

class `recsql.sqlarray.`**`KRingbuffer`**(*capacity*, **args*, ***kwargs*)

Ring buffer with key lookup.

Basically a ringbuffer for the keys and a dict (k,v) that is cleaned up to reflect the keys in the Ringbuffer.

`append`(*k*, *v*)

`clear`()

Reinitialize the KRingbuffer to empty.

class `recsql.sqlarray.`**`Ringbuffer`**(*capacity*, *iterable=None*)

Ring buffer of size capacity; 'pushes' data from left and discards on the right.

class `recsql.sqlarray.`**`SQLarray`**(*name=None*, *records=None*, *filename=None*, *columns=None*, *cachesize=5*, *connection=None*, *is_tmp=False*, ***kwargs*)

A SQL table that returns (mostly) rec arrays.

The *SQLarray* can be initialized from

1. an iterable of records (tuples), given in the *records* keyword argument, and the column names (provided in *columns*);
2. a string that contains a simple reStructured Text table (see *recsql.rest_table* for details);

3.a `numpy.recarray`.

Note: SQLite only understands standard Python types and hence has problems with many of the NumPy data types such as `numpy.int64`. When loading a recarray fails we try to convert all data types automatically to Python types (using *`recsql.convert.irecarray_to_py()`*). This might loose precision and/or even fail. It is also slow for larger arrays.

The class takes the following arguments:

`SQLarray`(*[name[, records[, columns[, cachesize=5, connection=None, dbfile=":memory:"]]]]*)

Arguments

name table name (can be referred to as '__self__' in SQL queries)

records numpy record array that describes the layout and initializes the table OR any iterable (and then columns must be set, too) OR a string that contains a single, *simple reStructured text table* (and the table name is set from the table name in the reST table.) If `None` then simply associate with existing table name.

filename Alternatively to *records*, read a reStructured table from *filename*.

columns sequence of column names (only used if records does not have attribute dtype.names) [`None`]

cachesize number of (query, result) pairs that are cached [5]

connection If not `None`, reuse this connection; this adds a new table to the same database, which allows more complicated queries with cross-joins. The table's connection is available as the attribute T.connection. [`None`]

dbfile Normally the db is held in memory (":memory:") but if a filename is provided then the underlying SQLite db is held on disk and can be accessed and restored (see *`SQLarray.save()`*). Only works with *connection* = `None` [":memory:"]

is_tmp `True`: create a tmp table; `False`: regular table in db [`False`]

Bugs

- `InterfaceError`: *Error binding parameter 0 - probably unsupported type*

  In this case the recarray contained types such as `numpy.int64` that are not understood by sqlite and which we were not able to convert to a Python type (using *`recsql.convert.irecarray_to_py()`*). Either convert the data manually (by setting the numpy dtypes yourself on the recarray, or better: feed a simple list of tuples ("records") to this class in *records*. Make sure that these tuples only contain standard Python types. Together with *records* you will also have to supply the names of the data columns in the keyword argument *columns*.

  If you are reading from a file then it might be simpler to use *`recsql.sqlarray.SQLarray_fromfile()`*.

`SELECT`(*fields, *args, **kwargs*)

Execute a simple SQL `SELECT` statement and returns values as new numpy rec array.

The arguments *fields* and the additional optional arguments are simply concatenated with additional SQL statements according to the template:

```
SELECT <fields> FROM __self__ [args]
```

The simplest fields argument is `"*"`.

Example: Create a recarray in which students with average grade less than 3 are listed:

```
result = T.SELECT("surname, subject, year, avg(grade) AS avg_grade",
                  "WHERE avg_grade < 3", "GROUP BY surname,subject",
                  "ORDER BY avg_grade,surname")
```

The resulting SQL would be:

```
SELECT surname, subject, year, avg(grade) AS avg_grade FROM __self__
     WHERE avg_grade < 3
     GROUP BY surname,subject
     ORDER BY avg_grade,surname
```

Note how one can use aggregate functions such avg().

The string '*__self__*' is automatically replaced with the table name (`T.name`); this can be used for cartesian products such as

```
LEFT JOIN __self__ WHERE ...
```

Note: See the documentation for *`sql()`* for more details on the available keyword arguments and the use of `?` parameter interpolation.

close()
Clean up (if no more connections to the db exist).

- For in-memory: Delete the underlying SQL table from the in-memory database.
- For on-disk: save and close connection

connection_count
Number of currently open connections to the database.

(Stored in table sqlarray_master.)

has_table(*name*)
Return `True` if the table *name* exists in the database.

limits(*variable*)
Return minimum and maximum of variable across all rows of data.

merge(*recarray*, *columns=None*)
Merge another recarray with the same columns into this table.

Arguments

recarray numpy record array that describes the layout and initializes the table

Returns n number of inserted rows

Raises Raises an exception if duplicate and incompatible data exist in the main table and the new one.

merge_table(*name*)
Merge an existing table in the database with the __self__ table.

Executes as `'INSERT INTO __self__ SELECT * FROM <name>'`. However, this method is probably used less often than the simpler *`merge()`*.

Arguments name name of the table in the database (must be compatible with __self__)

Returns n number of inserted rows

recarray
Return underlying SQL table as a read-only record array.

save()
Commit changes to file.

Only works if the SQLarray was created with they *dbfile* = `FILENAME` keyword. There is currently no way to save a in-memory db.

See also:

`aoft.DB.clone()`

selection(*SQL*, *parameters=None*, ***kwargs*)
Return a new SQLarray from a SELECT selection.

This method is useful to build complicated selections and essentially new tables from existing data. The result of the SQL query is stored as a new table in the database. By default, a unique name is created but this can be overridden with the *name* keyword.

Arguments

SQL SQL `SELECT` query string. A leading `SELECT * FROM __self__ WHERE` can be omitted (see examples below). The SQL is scrubbed and only data up to the first semicolon is used (note that this means that there cannot even be a semicolon in quotes; if this is a problem, file a bug report and it might be changed).

Keywords

name name of the table, `None` autogenerates a name unique to this query. *name* may not refer to the parent table itself. [`None`]

parameters tuple of values that are safely interpolated into subsequent `?` characters in the SQL string

force If `True` then an existing table of the same *name* is DROP``ped first.   If ``False and the table already exists then *SQL* is ignored and a *SQLarray* of the existing table *name* is returned. [`False`]

Returns a *SQLarray* referring to the table *name* in the database; it also inherits the `SQLarray.dbfile`

Examples:

```
s = SQLarray.selection('a > 3')
s = SQLarray.selection('a > ?', (3,))
s = SQLarray.selection('SELECT * FROM __self__ WHERE a > ? AND b < ?', (3, 10))
```

sql(*SQL*, *parameters=None*, *asrecarray=True*, *cache=True*)
Execute sql statement.

Arguments

SQL [string] Full SQL command; can contain the `?` place holder so that values supplied with the `parameters` keyword can be interpolated using the `pysqlite` interface.

parameters [tuple] Parameters for `?` interpolation.

asrecarray [boolean] `True`: return a `numpy.recarray` if possible; `False`: return records as a list of tuples. [`True`]

cache [boolean] Should the results be cached? Set to `False` for large queries to avoid memory issues. Queries with `?` place holders are never cached. [`True`]

Returns For *asrecarray* = `True` a `numpy.recarray` is returned; otherwise a simple list of tuples is returned.

Raises `TypeError` if the conversion to `recarray` fails for any reason.

Warning: There are **no sanity checks** applied to the SQL.

The last *cachesize* queries are cached (for *cache* = `True`) and are returned directly unless the table has been modified.

The string "__self__" in *SQL* is substituted with the table name. See the *SELECT()* method for more details.

sql_index (*index_name*, *column_names*, *unique=True*)
Add a named index on given columns to improve performance.

sql_select (*fields*, **args*, ***kwargs*)
Execute a simple SQL `SELECT` statement and returns values as new numpy rec array.

The arguments *fields* and the additional optional arguments are simply concatenated with additional SQL statements according to the template:

```
SELECT <fields> FROM __self__ [args]
```

The simplest fields argument is `"*"`.

Example: Create a recarray in which students with average grade less than 3 are listed:

```
result = T.SELECT("surname, subject, year, avg(grade) AS avg_grade",
               "WHERE avg_grade < 3", "GROUP BY surname,subject",
               "ORDER BY avg_grade,surname")
```

The resulting SQL would be:

```
SELECT surname, subject, year, avg(grade) AS avg_grade FROM __self__
     WHERE avg_grade < 3
     GROUP BY surname,subject
     ORDER BY avg_grade,surname
```

Note how one can use aggregate functions such avg().

The string '*__self__*' is automatically replaced with the table name (`T.name`); this can be used for cartesian products such as

```
LEFT JOIN __self__ WHERE ...
```

Note: See the documentation for *sql()* for more details on the available keyword arguments and the use of `?` parameter interpolation.

`recsql.sqlarray.`**`SQLarray_fromfile`** (*filename*, ***kwargs*)
Create a *SQLarray* from *filename*.

Uses the filename suffix to detect the contents:

rst, txt restructure text (see *recsql.rest_table*

csv comma-separated (see *recsql.csv_table*)

Arguments

filename name of the file that contains the data with the appropriate file extension

kwargs

- additional arguments for *SQLarray*
- additional arguments *recsql.csv_table.Table2array* or *recsql.rest_table.Table2array* such as *mode* or *autoncovert*.

2.4.2 `recsql.rest_table` — Parse a simple reST table

Turn a restructured text simple table into a numpy array. See the *Example* below for how the table must look like. The module allows inclusion of parameters and data in the documentation itself in a natural way. Thus the parameters are automatically documented and only exist in a single place. The idea is inspired by literate programming and is embodied by the DRY ("Do not repeat yourself") principle.

Limitations

Note that not the full specifications of the original restructured text simple table are supported. In order to keep the parser simple, the following additional restriction apply:

- All row data must be on a single line.
- Column spans are not supported.
- Headings must be single legal SQL and python words as they are used as column names.
- *Do not use TAB characters* to format the table but use spaces. (TABs do not have a well-defined width in number of spaces and will thus lead to wrongly parsed tables.)
- The delimiters are used to extract the fields. Only data within the range of the '=====' markers is used. Thus, each column marker *must* span the whole range of input. Otherwise, data will be lost.
- The keyword 'Table' must precede the first marker line and the table name must be provided in square brackets; the table name should be a valid SQL identifier.
- Currently, only a *single* table can be present in the string.
- Autoconversion of list fields might not always work...

Example

The following table is converted:

```
Table[laureates]: Physics Nobel prize statistics.
=============  ==========  =========
name           age         year
=============  ==========  =========
A. Einstein    42          1921
P. Dirac       31          1933
R. P. Feynman  47          1965
=============  ==========  =========
```

with

```
>>> import recsql.rest_table as T
>>> P = T.Table2array(T.__doc__)
>>> P.recarray()
rec.array([(u'A. Einstein', 42, 1921), (u'P. Dirac', 31, 1933),
       (u'R. P. Feynman', 47, 1965)],
    dtype=[('name', '<U52'), ('age', '<i4'), ('year', '<i4')])
```

Module content

The only class that the user really needs to know anything about is *recsql.rest_table.Table2array*.

class `recsql.rest_table.`**`Table2array`**(*string=None*, ***kwargs*)

Primitive parser that converts a simple reST table into `numpy.recarray`.

The table must be the only table in the text. It must look similar to the example below (variable parts in angle brackets, optional in double brackets, everything else must be there, matching is case sensitive, '....' signifies repetition in kind):

```
Table[<NAME>]: <<CAPTION>>
============  ===========  ======================  ....
<COLNAME 1>   <COLNAME 2>  ....                    ....
============  ===========  ======================  ....
<VALUE>       <VALUE>      <VALUE> <VALUE> ....
....
....
============  ===========  ======================  ....
```

Rows may *not* span multiple lines. The column names must be single words and legal python names (no spaces, no dots, not starting with a number).

Field values are converted to one of the following python types: *int*, *float*, or *str*.

If a value is quote with single or double quotation marks then the outermost quotation marks are stripped and the enclosed value treated as a string.

Note: Values such as 001 must be quoted as '001' or they will be interpreted as integers (1 in this case).

`__init__`(*string=None*, ***kwargs*)

Table2array(string) –> parser

Arguments

string string to be parsed

filename read from *filename* instead of string

autoconvert EXPERIMENTAL. `True`: replace certain values with special python values (see `convert.Autoconverter`) and possibly split values into lists (see *sep*). `False`: leave everything as it is (numbers as numbers and strings as strings).

mode mode of the `Autoconverter`

sep If set and *autoconvert* = `True` then split field values on the separator (using `split()`) before possible autoconversion. (NOT WORKING PROPERLY YET)

`recarray`()

Return a recarray from the (parsed) string.

exception `recsql.rest_table.`**`ParseError`**

Signifies a failure to parse.

2.4.3 `recsql.csv_table` — Parse a simple CSV table

Turn a CSV table into a numpy array.

Uses `csv` (requires python 2.6 or better).

class `recsql.csv_table.`**`Table2array`**(*filename=None*, *tablename='CSV'*, *encoding='utf-8'*, ***kwargs*)

Read a csv file and provide conversion to a `numpy.recarray`.

- Depending on the arguments, autoconversion of values can take place. See *`recsql.convert.Autoconverter`* for details.
- Table column headers are always read from the first row of the file.
- Empty rows are discarded.

`__init__`(*filename=None*, *tablename='CSV'*, *encoding='utf-8'*, ***kwargs*)

Initialize the class.

Arguments

filename CSV file (encoded with *encoding*)

name name of the table

autoconvert EXPERIMENTAL. `True`: replace certain values with special python values (see `convert.Autoconverter`) and possibly split values into lists (see *sep*). `False`: leave everything as it is (numbers as numbers and strings as strings).

mode mode of the `Autoconverter`

`recarray`()

Returns data as `numpy.recarray`.

`recsql.csv_table.`**`make_python_name`**(*s*, *default=None*, *number_prefix='N'*, *encoding='utf-8'*)

Returns a unicode string that can be used as a legal python identifier.

Arguments

s string

default use *default* if *s* is `None`

number_prefix string to prepend if *s* starts with a number

2.4.4 `recsql.convert` — converting entries of tables and arrays

class `recsql.convert.`**`Autoconverter`**(*mode='fancy'*, *mapping=None*, *active=True*, *sep=False*, ***kwargs*)

Automatically convert an input value to a special python object.

The *`Autoconverter.convert()`* method turns the value into a special python value and casts strings to the "best" type (see *`besttype()`*).

The defaults for the conversion of a input field value to a special python value are:

value	python
'—'	`None`
'none'	
'None'	
''	
'True'	`True`
'x'	
'X'	
'yes'	
'False'	`False`
'-'	
'no'	

If the *sep* keyword is set to a string instead of `False` then values are split into tuples. Probably the most convenient way to use this is to set *sep* = `True` (or `None`) because this splits on all white space whereas *sep* = ' ' would split multiple spaces.

Example

- With *sep* = `True`: 'foo bar 22 boing —' –> ('foo', 'boing', 22, None)
- With *sep* = ',': 1,2,3,4 –> (1,2,3,4)

`convert`(*x*)
Convert *x* (if in the active state)

`active`
If set to `True` then conversion takes place; `False` just returns `besttype()` applid to the value.

`__init__`(*mode='fancy'*, *mapping=None*, *active=True*, *sep=False*, ***kwargs*)
Initialize the converter.

Arguments

mode defines what the converter does

"simple" convert entries with `besttype()`

"singlet" convert entries with `besttype()` and apply mappings

"fancy" first splits fields into lists, tries mappings, and does the stuff that "singlet" does

"unicode" convert all entries with `to_unicode()`

mapping any dict-like mapping that supports lookup. If `None` then the hard-coded defaults are used

active* or *autoconvert

initial state of the `Autoconverter.active` toggle. `False` deactivates any conversion. [`True`]

sep character to split on (produces lists); use `True` or `None` (!) to split on all white space.

encoding encoding of the input data [utf-8]

percentify convert "34.4%" into 0.344 [`True`] `recsql.convert.`**`besttype`**`(`*x*, *encoding='utf-8'*, *percentify=True*`)`

Convert string x to the most useful type, i.e. int, float or unicode string.

If x is a quoted string (single or double quotes) then the quotes are stripped and the enclosed string returned. The string can contain any number of quotes, it is only important that it begins and ends with either single or double quotes.

percentify = `True` turns "34.4%" into the float 0.344.

Note: Strings will be returned as Unicode strings (using `unicode()`), based on the *encoding* argument, which is utf-8 by default.

`recsql.convert.`**`to_unicode`**`(`*obj*, *encoding='utf-8'*`)`

Convert obj to unicode (if it can be be converted)

from http://farmdev.com/talks/unicode/

`recsql.convert.`**`irecarray_to_py`**`(`*a*`)`

Slow conversion of a recarray into a list of records with python types.

Get the field names from `a.dtype.names`.

Returns iterator so that one can handle big input arrays

`recsql.convert.`**`pyify`**`(`*typestr*`)`

Return a Python `type` that most closely represents the type encoded by *typestr*

`recsql.convert.`**`to_pytypes`**`(`*a*`)`

Return array *a* with all types set to basic Python types.

`pyify()` is applied to all `dtypes` of the `numpy.ndarray` and then recast with `numpy.ndarray.astype()`. This approach can loose precision.

2.4.5 SQL support

`recsql.sqlfunctions` — Functions that enhance a SQLite db

This module contains new SQL functions to be added to a SQLite database that can be used in the same way as the builtin functions.

Example:

Add the functions to an existing connection in the following way (assuming that the db connection is available in `self.connection`):

```
from sqlfunctions import *
self.connection.create_function("sqrt", 1, _sqrt)
self.connection.create_function("sqr", 1, _sqr)
self.connection.create_function("periodic", 1, _periodic)
self.connection.create_function("pow", 2, _pow)
self.connection.create_function("match", 2, _match)   # implements MATCH
self.connection.create_function("regexp", 2, _regexp) # implements REGEXP
self.connection.create_function("fformat",2,_fformat)
self.connection.create_aggregate("std",1,_Stdev)
self.connection.create_aggregate("stdN",1,_StdevN)
self.connection.create_aggregate("median",1,_Median)
self.connection.create_aggregate("array",1,_NumpyArray)
self.connection.create_aggregate("histogram",4,_NumpyHistogram)
self.connection.create_aggregate("distribution",4,_NormedNumpyHistogram)
```

```
    self.connection.create_aggregate("meanhistogram",5,_MeanHistogram)
    self.connection.create_aggregate("stdhistogram",5,_StdHistogram)
    self.connection.create_aggregate("minhistogram",5,_MinHistogram)
    self.connection.create_aggregate("maxhistogram",5,_MaxHistogram)
    self.connection.create_aggregate("medianhistogram",5,_MedianHistogram)
    self.connection.create_aggregate("zscorehistogram",5,_ZscoreHistogram)
```

Module content

`recsql.sqlfunctions.`**`regularized_function`**(*x*, *y*, *func*, *bins=None*, *range=None*)

Compute func() over data aggregated in bins.

(x,y) –> (x', func(Y')) with Y' = {y: y(x) where x in x' bin}

First the data is collected in bins x' along x and then func is applied to all data points Y' that have been collected in the bin.

Arguments

x abscissa values (for binning)

y ordinate values (func is applied)

func a numpy ufunc that takes one argument, func(Y')

bins number or array

range limits (used with number of bins)

Returns

F,edges function and edges (midpoints = 0.5*(edges[:-1]+edges[1:]))

`sqlutils` – Helper functions

Helper functions that are used throughout the *`recsql`* package.

How to use the sql converters and adapters: Declare types as 'NumpyArray':

```
    cur.execute("CREATE TABLE test(a NumpyArray)")
    cur.execute("INSERT INTO test(a) values (?)", (my_array,))
```

or as column types:

```
    cur.execute('SELECT a as "a [NumpyArray]" from test')
```

Module content

class `recsql.sqlutil.`**`FakeRecArray`**(*iterable*, *columns*)

Pseudo recarray that is used to feed SQLarray:

Must only implement:

recarray.dtype.names sequence of column names iteration yield records

`recsql.sqlutil.`**`adapt_numpyarray`**(*a*)

adapter: store numpy arrays in the db as ascii pickles

`recsql.sqlutil.`**`adapt_object`**(*a*)

adapter: store python objects in the db as ascii pickles

`recsql.sqlutil.`**`convert_numpyarray`**(*s*)
converter: retrieve numpy arrays from the db as ascii pickles

`recsql.sqlutil.`**`convert_object`**(*s*)
convertor: retrieve python objects from the db as ascii pickles

2.5 Indices and tables

- genindex
- modindex
- search