PG Utils Documentation

Release 0.6.0

Jack Maney

October 14, 2016

Contents

When analyzing large datasets, it can often be useful to let the database do as much of the analysis as possible. While Pandas is great at manipulating datasets that are large enough to fit on one machine, but possibly not large enough to fit into memory, concerns over performance and data security can sometimes make analysis in the database more convenient.

This package is built for use with PostgreSQL. Support for other databases *might* follow (but don't hold your breath).

CHAPTER 1

Wait, what? Why?!?

Let's illustrate with a few examples. To begin with, making a connection is simple:

```
In [1]: from pg_utils import connection, table

In [2]: conn = connection.Connection()
```

The environment variables `pg_username`, `pg_password`, `pg_hostname`, and `pg_database` can be used to store values for the corresponding connection information. Of course, any of the username, password, hostname, or database can be overridden.

Moving on, let's build a simple table consisting of one million rows with one column chosen randomly and another sampled from the standard normal distribution (via the Box-Muller transform).

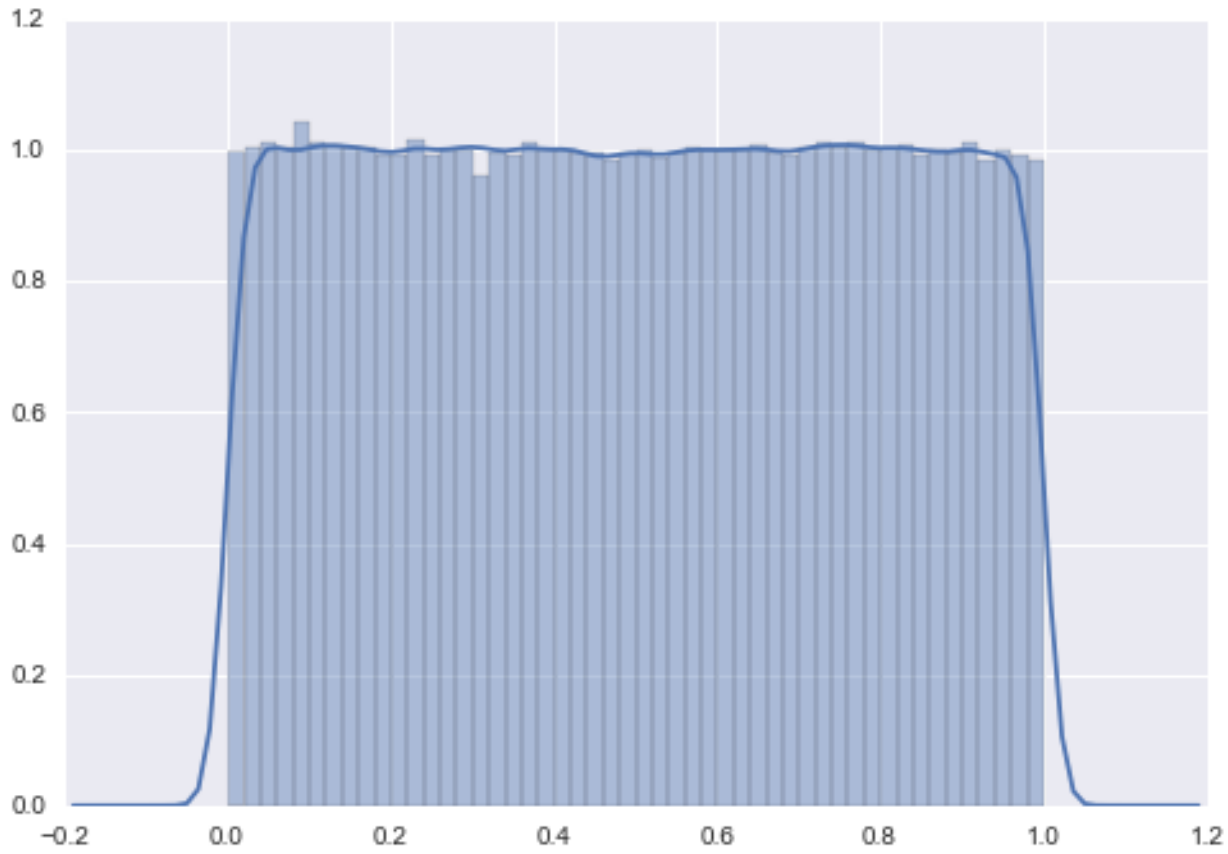
```
In [1]: from pg_utils import table

In [2]: t = table.Table.create("pg_utils_test",
   ...:          """create table pg_utils_test as
                     select random() as x,
   ...:              sqrt(-2 * ln(u1))*cos(2*PI()*u2) as y
   ...:              from(
   ...:                  select random() as u1, random() as u2
   ...:                  from generate_series(1, 1000000)
   ...:              )a""")
```

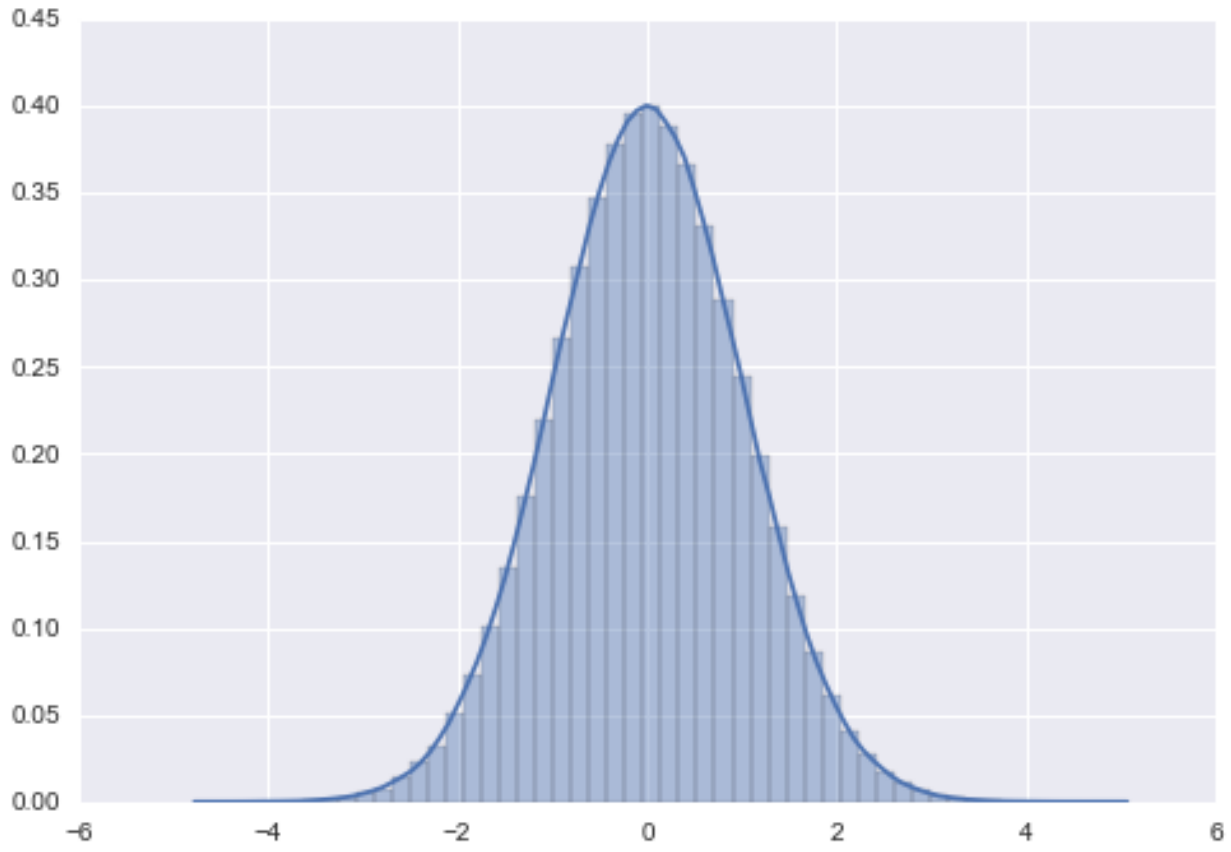
The object `t` is a metadata object. It doesn't hold any of the actual data within the table. However, we have a limited subset of the Pandas API that works via the database. For example:

```
In [3]: t.describe()
Out[3]:
                      x                y
count    1000000.000000  1000000.000000
mean           0.499628       -0.000075
std_dev        0.288671        0.999630
minimum        0.000001       -4.589661
25%            0.249420       -0.672603
50%            0.499709        0.000695
75%            0.749733        0.673413
maximum        0.999999        4.867347
```

Note that *none* of those calculations are done locally. They're all done in the database.

Also, `t.x.distplot(bins=1000)` and `t.y.distplot(bins=1000)` produce the following histograms (with KDEs):


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2


0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−6
−4
−2
0
2
4
6

CHAPTER 2

Note

The `Table` API has significantly changed from `0.3.x` to `0.4.0`. In particular, schema and connection parameters are now optional (replaced with your username and a fresh connection, respectively). This leaves the table name as the only required parameter for many of these methods. If `table_name` is already qualified with a schema (eg `"foo.bar"`), then pg-utils will Do The Right Thing and just set `schema="foo"` and `table_name="bar"`.

CHAPTER 3

Goals

The goals for this package are:

- Providing a simple `Connection` object that builds easy connections to the database based off of environment variables (overridden with parameters, if specified).
- Mocking Pandas Series and DataFrame objects with metadata constructs of `Column` and `Table` (respectively). Columns and Tables will implement some parts of the Pandas API that do calculations in the database.
- Possibly other tools for automation of simple tasks and conveniently displaying metadata (if needed).

CHAPTER 4

Non-Goals

This package will never be:

- An object-relational mapper (ORM). The only SQL-ish operations will be those that are implemented and that mock SQL-ish bits of the Pandas API for Series and/or DataFrames.

CHAPTER 5

Installation

It's up on PyPI. So, just do

```
pip install pg-utils
```

for the base package, or

```
pip install pg-utils[graphics]
```

to install Seaborn for graphical visualizations.

CHAPTER 6

Relevant pieces of the API

6.1 Connection

class pg_utils.connection.**Connection**(*username=None*, *password=None*, *hostname=None*, *database=None*, *env_username='pg_username'*, *env_password='pg_password'*, *env_hostname='pg_hostname'*, *env_database='pg_database'*, ***other_connection_kwargs*)

This is a wrapper class around psycopg2's `Connection` object. Its main purpose is for the simple specification of login information via environment variables.

Parameters

- **username** (*str*) – Username. Overrides the corresponding environment variable.
- **password** (*str*) – Password. Overrides the corresponding environment variable.
- **hostname** (*str*) – Hostname. Overrides the corresponding environment variable.
- **database** (*str*) – The name of the database. Overrides the corresponding environment variable.
- **env_username** (*str*) – The name of the environment variable to use for your username.
- **env_password** (*str*) – The name of the environment variable to use for your password.
- **env_hostname** (*str*) – The name of the environment variable to use for the hostname.
- **env_database** (*str*) – The name of the environment variable to use for the database.
- **other_connection_kwargs** (*None|dict*) – Other keyword arguments (if any) that you'd like to pass to the psycopg2 `Connection` object.

Variables **connection** (*psycopg2.extensions.connection*) – The resulting raw connection object.

close()
A simple wrapper around the `close` method of the `connection` attribute.

rollback()
A simple wrapper around the `rollback` method of the `connection` attribute.

commit()
A simple wrapper around the `commit` method of the `connection` attribute.

cursor(**args*, ***kwargs*)
A simple wrapper around the `cursor` factory method of the `connection` attribute.

6.2 Columns

class `pg_utils.column.base.`**Column**(*name*, *parent_table*)
In Pandas, a column of a DataFrame is represented as a Series.

Similarly, a column in a database table is represented by an object from this class.

Note that the Series represented by these columns have the default index (ie non-negative, consecutive integers starting at zero). Thus, for the portion of the Pandas Series API mocked here, we need not worry about multilevel (hierarchical) indices.

Parameters

- **name** (*str*) – The name of the column. Required.
- **parent_table** (*pg_utils.table.Table*) – The table to which this column belongs. Required.

select_all_query()
Provides the SQL used when selecting everything from this column.

Returns The SQL statement.

Return type str

sort_values(*ascending=True*, *limit=None*, ***sql_kwargs*)
Mimics the method pandas.Series.sort_values.

Parameters

- **limit** (*int | None*) – Either a positive integer for the number of rows to take or `None` to take all.
- **ascending** (*bool*) – Sort ascending vs descending.
- **sql_kwargs** (*dict*) – A dictionary of keyword arguments passed into pandas.read_sql.

Returns The resulting series.

Return type pandas.Series

unique()
Returns an array of unique values in this column. Includes `null` (represented as `None`). :return: The unique values. :rtype: np.array

head(*num_rows=10*)
Fetches some values of this column.

Parameters **num_rows** (*int | str*) – Either a positive integer number of values or the string *"all"* to fetch all values

Returns A NumPy array of the values

Return type np.array

is_unique
Determines whether or not the values of this column are all unique (ie whether this column is a unique identifier for the table). :return: Whether or not this column contains unique values. :rtype: bool

dtype
The `dtype` of this column (represented as a string).

Returns The `dtype`.

Return type str

describe (*percentiles=None*, *type_='continuous'*)
This mocks the method *pandas.Series.describe*, and provides a series with the same data (just calculated by the database).

Parameters

- **percentiles** (*None|list[float]*) – A list of percentiles to evaluate (with numbers between 0 and 1). If not specified, quartiles (0.25, 0.5, 0.75) are used.
- **type** (*str*) – Specifies whether the percentiles are to be taken as discrete or continuous. Must be one of *"discrete"* or *"continuous"*.

Returns A series returning the description of the column, in the same format as `pandas.Series.describe`.

Return type pandas.Series

distplot (**args*, ***kwargs*)
Produces a `distplot`. See the seaborn docs on `distplot` for more information.

Note that this requires Seaborn in order to function.

Parameters

- **bins** (*int|None*) – The number of bins to use. If unspecified, the Freedman-Diaconis rule will be used to determine the number of bins.
- **kwargs** (*dict*) – A dictionary of options to pass on to seaborn.distplot.

values
Mocks the method *pandas.Series.values*, returning a simple NumPy array consisting of the values of this column.

Returns The NumPy array containing the values.

Return type np.array

mean
Mocks the `pandas.Series.mean` method to give the mean of the values in this column. :return: The mean. :rtype: float

max
Mocks the `pandas.Series.max` method to give the maximum of the values in this column. :return: The maximum. :rtype: float

min
Mocks the `pandas.Series.min` method to give the maximum of the values in this column. :return: The minimum. :rtype: float

size
Mocks the `pandas.Series.size` property to give a count of the values in this column. :return: The count. :rtype: int

6.3 Tables

class pg_utils.table.table.**Table**(**args*, ***kwargs*)

This class is used for representing table metadata.

Variables

- **conn** (pg_utils.connection.Connection) – A connection to be used by this table.
- ***name*** (*str*) – The fully-qualified name of this table.
- **column_names** (*tuple[str]*) – A list of column names for the table, as found in the database.
- **columns** (*tuple[Column]*) – A tuple of Column objects.
- ***numeric_columns*** (*tuple[str]*) – A list of column names corresponding to the column_names in the table that have some kind of number datatype (int, float8, numeric, etc).

Parameters

- **table_name** (*str*) – The name of the table in the database. If it's qualified with a schema, then leave the schema argument alone.
- **schema** (*None|str*) – The name of the schema in which this table lies. If unspecified and the value of table_name doesn't include a schema, then the (OS-specified) username of the given user is taken to be the schema.
- **conn** (*None|pg_utils.connection.Connection*) – A connection object that's used to fetch data and metadata. If not specified, a new connection is made with default arguments provided for username, password, etc.
- **columns** (*str|list[str]|tuple[str]*) – An iterable of specified column names. It's used by the __getitem__ magic method, so you shouldn't need to fiddle with this.
- **check_existence** (*bool*) – If enabled, an extra check is made to ensure that the table referenced by this object actually exists in the database.
- **debug** (*bool*) – Enable to get some extra logging that's useful for debugging stuff.

classmethod create(**args*, ***kwargs*)

This is the constructor that's easiest to use when creating a new table.

Parameters

- **table_name** (*str*) – As mentioned above.
- **create_stmt** (*str*) – A string of SQL (presumably including a "CREATE TABLE" statement for the corresponding database table) that will be executed before __init__ is run.

Note: The statement drop table if exists schema.table_name; is executed **before** the SQL in create_stmt is executed.

Parameters

- **conn** (*None|pg_utils.connection.Connection*) – A Connection object to use for creating the table. If not specified, a new connection will be created with no arguments. Look at the docs for the Connection object for more information.

- **schema** (*None | str*) – A specified schema (optional).
- **args** – Other positional arguments to pass to the initializer.
- **kwargs** – Other keyword arguments to pass to the initializer.

Returns The corresponding `Table` object *after* the `create_stmt` is executed.

classmethod from_table(*table*, **args*, ***kwargs*)
This class method constructs a table from a given table. Used to give a fresh `Table` object with different columns, but all other parameters the same as the given table.

If the `columns` attribute only specifies one column, then a `Column` object will be returned. :param Table table: The table object from which the output will be created. :param list args: Any positional arguments (if any). :param dict kwargs: Any keyword arguments to pass along (if any). :return: Either a fresh `Table` or `Column`, depending on whether the `columns` parameter is restricted to just a single column. :rtype: Column|Table

count
Returns the number of rows in the corresponding database table.

head(*num_rows=10*, ***read_sql_kwargs*)
Returns some of the rows, returning a corresponding Pandas DataFrame.

Parameters

- **num_rows** (*int | str*) – The number of rows to fetch, or `"all"` to fetch all of the rows.
- **read_sql_kwargs** (*dict*) – Any other keyword arguments that you'd like to pass into `pandas.read_sql` (as documented here).

Returns The resulting data frame.

Return type pandas.core.frame.DataFrame

shape
As in the property of Pandas DataFrames by the same name, this gives a tuple showing the dimensions of the table: `(number of rows, number of columns)`

dtypes
Mimics the *pandas.DataFrame.dtypes* property, giving a Series of dtypes (given as strings) corresponding to each column. :return: The Series of dtypes. :rtype: pd.Series

insert(*row*, *columns=None*)
Inserts a single tuple into the table.

Parameters

- **row** (*list | pandas.Series*) – A list or Series of items to insert. If a list, its length must match up with the number of columns that we'll insert. If it's a series, the column names must be contained within the index.
- **columns** (*None | list[str] | tuple[str]*) – An iterable of column names to use, that must be contained within this table. If not specified, all of the columns are taken.

Returns Returns a boolean indicating success.

Return type bool

insert_csv(*file_name*, *columns=None*, *header=True*, *sep=','*, *null=''*, *size=8192*)
A wrapper around the copy_expert method of the psycopg2 cursor class to do a bulk insert into the table.

Parameters

- **file_name** (*str*) – The name of the CSV file.

- **columns** (*None|list[str]|tuple[str]*) – An iterable of column names to use, that must be contained within this table. If not specified, all of the columns are taken.
- **header** (*bool*) – Indicates whether or not the file has a header.
- **sep** (*str*) – The separator character.
- **null** (*str*) – The string used to indicate null values.
- **size** (*int*) – The size of the buffer that `psycopg2.cursor.copy_expert` uses.

insert_dataframe(*data_frame*, *encoding='utf8'*, ***csv_kwargs*)

Does a bulk insert of a given pandas DataFrame, writing it to a (temp) CSV file, and then importing it.

Parameters

- **data_frame** (*pd.DataFrame*) – The DataFrame that is to be inserted into this table.
- **encoding** (*str*) – The encoding of the CSV file.
- **csv_kwargs** – Other keyword arguments that are passed to the `insert_csv` method.

sort_values(*by*, *ascending=True*, ***sql_kwargs*)

Mimicks the pandas.DataFrame.sort_values method.

Parameters

- **by** (*str|list[str]*) – A string or list of strings representing one or more column names by which to sort.
- **ascending** (*bool|list[bool]*) – Whether to sort ascending or descending. This must match the number of columns by which we're sorting, although if it's just a single value, it'll be used for all columns.
- **sql_kwargs** (*dict*) – A dictionary of keyword arguments passed into pandas.read_sql.

Returns Values of the sorted DataFrame.

Return type pandas.DataFrame

describe(*columns=None*, *percentiles=None*, *type_='continuous'*)

Mimics the `pandas.DataFrame.describe` method, getting basic statistics of each numeric column.

Parameters

- **columns** (*None|list[str]*) – A list of column names to which the description should be restricted. If not specified, then all numeric columns will be included.
- **percentiles** (*list[float]|None*) – A list of percentiles (given as numbers between 0 and 1) to compute. If not specified, quartiles will be used (ie 0.25, 0.5, 0.75).
- **type** (*str*) – Specifies whether the percentiles are to be taken as discrete or continuous. Must be one of *"discrete"* or *"continuous"*.

Returns A series representing the statistical description for each column. The format is the same as the output of `pandas.DataFrame.describe`.

Return type pd.DataFrame

pairplot(**args*, ***kwargs*)

Yields a Seaborn pairplot for all of the columns of this table that are of a numeric datatype.

Parameters **kwargs** (*dict*) – Optional keyword arguments to pass into seaborn.pairplot.

Returns The grid of plots.

numeric_columns
A tuple of names belonging to columns that have a numeric datatype.

numeric_array_columns
A tuple of names belonging to columns that are an array of a numeric datatype (eg `int[]`, `double precision[]`, etc).

column_data_types
A dictionary mapping column names to their corresponding datatypes.

schema
The name of the schema.

table_name
The name of the table (without the schema included).

name
The fully-qualified name of the table.

drop()
Drops the table and deletes this object (by calling `del` on it).

static **exists**(**args*, ***kwargs*)
A static method that returns whether or not the given table exists.

Parameters

- **table_name** (*str*) – The name of the table.
- **schema** (*None|str*) – The name of the schema (or current username if not provided).
- **conn** (*None|pg_utils.connection.Connection*) – A connection to the database. If not provided, a new connection is created with default arguments.

Returns Whether or not the table exists.

Return type bool

6.4 Miscellaneous Utility Functions

This module just contains utility functions that don't directly fit anywhere else. You shouldn't need to tinker with these.

pg_utils.util.**position_of_positional_arg**(*arg_name*, *fcn*)
Finds the index of a function's named positional arguments that has a specific name. For example:

```
In [1]: def foo(x, y, z, w=1, q=None): pass

In [2]: _position_of_positional_arg("y", foo)
Out[2]: 1

In [3]: _position_of_positional_arg("z", foo)
Out[3]: 2

In [4]: _position_of_positional_arg("w", foo)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<...snip...>

ValueError: Argument 'w' not found as a named positional argument of function foo
```

Parameters

- **arg_name** (*str*) – The name of the parameter to search for.
- **fcn** – A function whose parameters we will search.

Returns The index (if it exists).

Return type int

Raises ValueError: If `arg_name` isn't the name of any named positional argument of `fcn`.

pg_utils.util.**process_schema_and_conn**(*f*)

This decorator does the following (which is needed a few times in the `Table` class):

- Standardizes the `table_name` and `schema` parameters (the first of which must be positional and named, and the latter of which must be a keyword parameter). In particular, if `table_name == "foo.bar"`, then `schema` is replaced with `"foo"` and `table_name` with `"bar"`. If the table name is not qualified with a schema, then both `schema` and `table_name` are left alone.
- If no `conn` keyword argument is given (or if it's `None`), then that keyword argument is replaced with a fresh `Connection` object.

Raises

- **ValueError** – if mismatching schemas are found in both `schema` and `table_name`
- **ValueError** – if we can't parse the table name (ie more than one occurrence of `"."`).

pg_utils.util.**seaborn_required**(*f*)

This decorator makes sure that seaborn is imported before the function is run.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

p

C

D

E

F

H

I

M

N

P

R

S

T

U

V