dataset Documentation

Release 1.0.0

Friedrich Lindenberg, Gregor Aisch, Stefan Wehrmeyer

Jan 13, 2018

Contents

Although managing data in relational database has plenty of benefits, they're rarely used in day-to-day work with small to medium scale datasets. But why is that? Why do we see an awful lot of data stored in static files in CSV or JSON format, even though they are hard to query and update incrementally?

The answer is that **programmers are lazy**, and thus they tend to prefer the easiest solution they find. And in **Python**, a database isn't the simplest solution for storing a bunch of structured data. This is what **dataset** is going to change!

dataset provides a simple abstraction layer removes most direct SQL statements without the necessity for a full ORM model - essentially, databases can be used like a JSON file or NoSQL store.

A simple data loading script using **dataset** might look like this:

```
import dataset

db = dataset.connect('sqlite:///:memory:')

table = db['sometable']
table.insert(dict(name='John Doe', age=37))
table.insert(dict(name='Jane Doe', age=34, gender='female'))

john = table.find_one(name='John Doe')
```

Here is similar code, without dataset.

CHAPTER 1

Features

- **Automatic schema**: If a table or column is written that does not exist in the database, it will be created automatically.
- **Upserts**: Records are either created or updated, depending on whether an existing version can be found.
- **Query helpers** for simple queries such as `all` rows in a table or all `distinct` values across a set of columns.
- **Compatibility**: Being built on top of SQLAlchemy, `dataset` works with all major databases, such as SQLite, PostgreSQL and MySQL.

CHAPTER 2

Contents

2.1 Installation Guide

The easiest way is to install `dataset` from the Python Package Index using `pip` or `easy_install`:

```
$ pip install dataset
```

To install it manually simply download the repository from Github:

```
$ git clone git://github.com/pudo/dataset.git
$ cd dataset/
$ python setup.py install
```

Depending on the type of database backend, you may also need to install a database specific driver package. For MySQL, this is `MySQLdb`, for Postgres its `psycopg2`. SQLite support is integrated into Python.

2.2 Quickstart

Hi, welcome to the twelve-minute quick-start tutorial.

2.2.1 Connecting to a database

At first you need to import the dataset package :)

```
import dataset
```

To connect to a database you need to identify it by its URL, which basically is a string of the form `"dialect://user:password@host/dbname"`. Here are a few examples for different database backends:

```
# connecting to a SQLite database
db = dataset.connect('sqlite:///mydatabase.db')

# connecting to a MySQL database with user and password
db = dataset.connect('mysql://user:password@localhost/mydatabase')

# connecting to a PostgreSQL database
db = dataset.connect('postgresql://scott:tiger@localhost:5432/mydatabase')
```

It is also possible to define the *URL* as an environment variable called *DATABASE_URL* so you can initialize database connection without explicitly passing an *URL*:

```
db = dataset.connect()
```

Depending on which database you're using, you may also have to install the database bindings to support that database. SQLite is included in the Python core, but PostgreSQL requires `psycopg2` to be installed. MySQL can be enabled by installing the `mysql-db` drivers.

2.2.2 Storing data

To store some data you need to get a reference to a table. You don't need to worry about whether the table already exists or not, since dataset will create it automatically:

```
# get a reference to the table 'user'
table = db['user']
```

Now storing data in a table is a matter of a single function call. Just pass a dict to *insert*. Note that you don't need to create the columns *name* and *age* – dataset will do this automatically:

```
# Insert a new record.
table.insert(dict(name='John Doe', age=46, country='China'))

# dataset will create "missing" columns any time you insert a dict with an unknown key
table.insert(dict(name='Jane Doe', age=37, country='France', gender='female'))
```

Updating existing entries is easy, too:

```
table.update(dict(name='John Doe', age=47), ['name'])
```

The list of filter columns given as the second argument filter using the values in the first column. If you don't want to update over a particular value, just use the auto-generated `id` column.

2.2.3 Using Transactions

You can group a set of database updates in a transaction. In that case, all updates are committed at once or, in case of exception, all of them are reverted. Transactions are supported through a context manager, so they can be used through a `with` statement:

```
with dataset.connect() as tx:
    tx['user'].insert(dict(name='John Doe', age=46, country='China'))
```

You can get same functionality by invoking the methods `begin()`, `commit()` and `rollback()` explicitly:

```
db = dataset.connect()
db.begin()
try:
    db['user'].insert(dict(name='John Doe', age=46, country='China'))
    db.commit()
except:
    db.rollback()
```

Nested transactions are supported too:

```
db = dataset.connect()
with db as tx1:
    tx1['user'].insert(dict(name='John Doe', age=46, country='China'))
    with db as tx2:
        tx2['user'].insert(dict(name='Jane Doe', age=37, country='France', gender=
↪'female'))
```

2.2.4 Inspecting databases and tables

When dealing with unknown databases we might want to check their structure first. To start exploring, let's find out what tables are stored in the database:

```
>>> print(db.tables)
[u'user']
```

Now, let's list all columns available in the table `user`:

```
>>> print(db['user'].columns)
[u'id', u'country', u'age', u'name', u'gender']
```

Using `len()` we can get the total number of rows in a table:

```
>>> print(len(db['user']))
2
```

2.2.5 Reading data from tables

Now let's get some real data out of the table:

```
users = db['user'].all()
```

If we simply want to iterate over all rows in a table, we can omit *`all()`*:

```
for user in db['user']:
   print(user['age'])
```

We can search for specific entries using *`find()`* and *`find_one()`*:

```
# All users from China
chinese_users = table.find(country='China')

# Get a specific user
john = table.find_one(name='John Doe')
```

```
# Find by comparison
elderly_users = table.find(table.table.columns.age >= 70)
```

Using *distinct()* we can grab a set of rows with unique values in one or more columns:

```
# Get one user per country
db['user'].distinct('country')
```

Finally, you can use the `row_type` parameter to choose the data type in which results will be returned:

```
import dataset
from stuf import stuf

db = dataset.connect('sqlite:///mydatabase.db', row_type=stuf)
```

Now contents will be returned in `stuf` objects (basically, `dict` objects whose elements can be acessed as attributes (`item.name`) as well as by index (`item['name']`).

2.2.6 Running custom SQL queries

Of course the main reason you're using a database is that you want to use the full power of SQL queries. Here's how you run them with `dataset`:

```
result = db.query('SELECT country, COUNT(*) c FROM user GROUP BY country')
for row in result:
   print(row['country'], row['c'])
```

The `query()` method can also be used to access the underlying SQLAlchemy core API, which allows for the programmatic construction of more complex queries:

```
table = db['user'].table
statement = table.select(table.c.name.like('%John%'))
result = db.query(statement)
```

2.3 API documentation

2.3.1 Connecting

`dataset.`**`connect`**(*url=None*, *schema=None*, *reflect_metadata=True*, *engine_kwargs=None*, *reflect_views=True*, *ensure_schema=True*, *row_type=<class 'collections.OrderedDict'>*)

Opens a new connection to a database.

url can be any valid SQLAlchemy engine URL. If *url* is not defined it will try to use *DATABASE_URL* from environment variable. Returns an instance of *Database*. Set *reflect_metadata* to False if you don't want the entire database schema to be pre-loaded. This significantly speeds up connecting to large databases with lots of tables. *reflect_views* can be set to False if you don't want views to be loaded. Additionally, *engine_kwargs* will be directly passed to SQLAlchemy, e.g. set *engine_kwargs={'pool_recycle': 3600}* will avoid DB connection timeout. Set *row_type* to an alternate dict-like class to change the type of container rows are stored in.:

```
db = dataset.connect('sqlite:///factbook.db')
```

2.3.2 Notes

- **dataset** uses SQLAlchemy connection pooling when connecting to the database. There is no way of explicitly clearing or shutting down the connections, other than having the dataset instance garbage collected.

2.3.3 Database

class dataset.**Database**(*url*, *schema=None*, *reflect_metadata=True*, *engine_kwargs=None*, *reflect_views=True*, *ensure_schema=True*, *row_type=<class 'collections.OrderedDict'>*)

A database object represents a SQL database with multiple tables.

begin()

Enter a transaction explicitly.

No data will be written until the transaction has been committed.

commit()

Commit the current transaction.

Make all statements executed since the transaction was begun permanent.

create_table(*table_name*, *primary_id=None*, *primary_type=None*)

Create a new table.

Either loads a table or creates it if it doesn't exist yet. You can define the name and type of the primary key field, if a new table is to be created. The default is to create an auto-incrementing integer, `id`. You can also set the primary key to be a string or big integer. The caller will be responsible for the uniqueness of `primary_id` if it is defined as a text type.

Returns a *Table* instance.

```
table = db.create_table('population')

# custom id and type
table2 = db.create_table('population2', 'age')
table3 = db.create_table('population3',
                         primary_id='city',
                         primary_type=db.types.text)
# custom length of String
table4 = db.create_table('population4',
                         primary_id='city',
                         primary_type=db.types.string(25))
# no primary key
table5 = db.create_table('population5',
                         primary_id=False)
```

get_table(*table_name*, *primary_id=None*, *primary_type=None*)

Load or create a table.

This is now the same as `create_table`.

```
table = db.get_table('population')
# you can also use the short-hand syntax:
table = db['population']
```

load_table(*table_name*)

Load a table.

This will fail if the tables does not already exist in the database. If the table exists, its columns will be reflected and are available on the *Table* object.

Returns a *Table* instance.

```
table = db.load_table('population')
```

query(*query*, **args*, ***kwargs*)
Run a statement on the database directly.

Allows for the execution of arbitrary read/write queries. A query can either be a plain text string, or a SQLAlchemy expression. If a plain string is passed in, it will be converted to an expression automatically.

Further positional and keyword arguments will be used for parameter binding. To include a positional argument in your query, use question marks in the query (i.e. `SELECT * FROM tbl WHERE a = ?`). For keyword arguments, use a bind parameter (i.e. `SELECT * FROM tbl WHERE a = :foo`).

```
statement = 'SELECT user, COUNT(*) c FROM photos GROUP BY user'
for row in db.query(statement):
    print(row['user'], row['c'])
```

The returned iterator will yield each result sequentially.

rollback()
Roll back the current transaction.

Discard all statements executed since the transaction was begun.

tables
Get a listing of all tables that exist in the database.

2.3.4 Table

class dataset.**Table**(*database*, *table_name*, *primary_id=None*, *primary_type=None*, *auto_create=False*)
Represents a table in a database and exposes common operations.

all(**_clauses*, ***kwargs*)
Perform a simple search on the table.

Simply pass keyword arguments as `filter`.

```
results = table.find(country='France')
results = table.find(country='France', year=1980)
```

Using `_limit`:

```
# just return the first 10 rows
results = table.find(country='France', _limit=10)
```

You can sort the results by single or multiple columns. Append a minus sign to the column name for descending order:

```
# sort results by a column 'year'
results = table.find(country='France', order_by='year')
# return all rows sorted by multiple columns (descending by year)
results = table.find(order_by=['country', '-year'])
```

To perform complex queries with advanced filters or to perform aggregation, use *db.query()* instead.

columns
Get a listing of all columns that exist in the table.

count(**_clauses*, ***kwargs*)
Return the count of results for the given filter set.

create_column(*name*, *type*)
Create a new column `name` of a specified type.

```
table.create_column('created_at', db.types.datetime)
```

create_index(*columns*, *name=None*, ***kw*)
Create an index to speed up queries on a table.

If no `name` is given a random name is created.

```
table.create_index(['name', 'country'])
```

delete(**clauses*, ***filters*)
Delete rows from the table.

Keyword arguments can be used to add column-based filters. The filter criterion will always be equality:

```
table.delete(place='Berlin')
```

If no arguments are given, all records are deleted.

distinct(**args*, ***_filter*)
Return all the unique (distinct) values for the given `columns`.

```
# returns only one row per year, ignoring the rest
table.distinct('year')
# works with multiple columns, too
table.distinct('year', 'country')
# you can also combine this with a filter
table.distinct('year', country='China')
```

drop()
Drop the table from the database.

Deletes both the schema and all the contents within it.

drop_column(*name*)
Drop the column `name`.

```
table.drop_column('created_at')
```

find(**_clauses*, ***kwargs*)
Perform a simple search on the table.

Simply pass keyword arguments as `filter`.

```
results = table.find(country='France')
results = table.find(country='France', year=1980)
```

Using `_limit`:

```
# just return the first 10 rows
results = table.find(country='France', _limit=10)
```

You can sort the results by single or multiple columns. Append a minus sign to the column name for descending order:

```
# sort results by a column 'year'
results = table.find(country='France', order_by='year')
# return all rows sorted by multiple columns (descending by year)
results = table.find(order_by=['country', '-year'])
```

To perform complex queries with advanced filters or to perform aggregation, use *db.query()* instead.

find_one(**args*, ***kwargs*)
Get a single result from the table.

Works just like *find()* but returns one result, or `None`.

```
row = table.find_one(country='United States')
```

insert(*row*, *ensure=None*, *types=None*)
Add a `row` dict by inserting it into the table.

If `ensure` is set, any of the keys of the row are not table columns, they will be created automatically.

During column creation, `types` will be checked for a key matching the name of a column to be created, and the given SQLAlchemy column type will be used. Otherwise, the type is guessed from the row value, defaulting to a simple unicode field.

```
data = dict(title='I am a banana!')
table.insert(data)
```

Returns the inserted row's primary key.

insert_ignore(*row*, *keys*, *ensure=None*, *types=None*)
Add a `row` dict into the table if the row does not exist.

If rows with matching `keys` exist they will be added to the table.

Setting `ensure` results in automatically creating missing columns, i.e., keys of the row are not table columns.

During column creation, `types` will be checked for a key matching the name of a column to be created, and the given SQLAlchemy column type will be used. Otherwise, the type is guessed from the row value, defaulting to a simple unicode field.

```
data = dict(id=10, title='I am a banana!')
table.insert_ignore(data, ['id'])
```

insert_many(*rows*, *chunk_size=1000*, *ensure=None*, *types=None*)
Add many rows at a time.

This is significantly faster than adding them one by one. Per default the rows are processed in chunks of 1000 per commit, unless you specify a different `chunk_size`.

See *insert()* for details on the other parameters.

```
rows = [dict(name='Dolly')] * 10000
table.insert_many(rows)
```

update(*row*, *keys*, *ensure=None*, *types=None*, *return_count=False*)
Update a row in the table.

The update is managed via the set of column names stated in `keys`: they will be used as filters for the data to be updated, using the values in `row`.

```
# update all entries with id matching 10, setting their title columns
data = dict(id=10, title='I am a banana!')
table.update(data, ['id'])
```

If keys in `row` update columns not present in the table, they will be created based on the settings of `ensure` and `types`, matching the behavior of *insert()*.

upsert(*row*, *keys*, *ensure=None*, *types=None*)

An UPSERT is a smart combination of insert and update.

If rows with matching `keys` exist they will be updated, otherwise a new row is inserted in the table.

```
data = dict(id=10, title='I am a banana!')
table.upsert(data, ['id'])
```

2.3.5 Data Export

Note: Data exporting has been extracted into a stand-alone package, datafreeze. See the relevant repository here.

datafreeze.**freeze**(*result*, *format='csv'*, *filename='freeze.csv'*, *fileobj=None*, *prefix='.'*, *mode='list'*, ***kw*)

Perform a data export of a given result set. This is a very flexible exporter, allowing for various output formats, metadata assignment, and file name templating to dump each record (or a set of records) into individual files.

```
result = db['person'].all()
dataset.freeze(result, format='json', filename='all-persons.json')
```

Instead of passing in the file name, you can also pass a file object:

```
result = db['person'].all()
fh = open('/dev/null', 'wb')
dataset.freeze(result, format='json', fileobj=fh)
```

Be aware that this will disable file name templating and store all results to the same file.

If `result` is a table (rather than a result set), all records in the table are exported (as if `result.all()` had been called).

freeze supports two values for `mode`:

***list* (default)** The entire result set is dumped into a single file.

item One file is created for each row in the result set.

You should set a `filename` for the exported file(s). If `mode` is set to *item* the function would generate one file per row. In that case you can use values as placeholders in filenames:

```
dataset.freeze(res, mode='item', format='json',
               filename='item-{{id}}.json')
```

The following output `format` s are supported:

csv Comma-separated values, first line contains column names.

json A JSON file containing a list of dictionaries for each row in the table. If a `callback` is given, JSON with padding (JSONP) will be generated.

tabson Tabson is a smart combination of the space-efficiency of the CSV and the parsability and structure of JSON.

You can pass additional named parameters specific to the used format.

As an example, you can freeze to minified JSON with the following:

dataset.freeze(res, format='json', indent=4, wrap=False, filename='output.json')

json* and *tabson

***callback*:** if provided, generate a JSONP string using the given callback function, i.e. something like *callback && callback({...})*

***indent*:** if *indent* is a non-negative integer (it is `2` by default when you call *dataset.freeze*, and `None` via the `datafreeze` command), then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. `None` is the most compact representation.

***meta*:** if *meta* is not `None` (default: `{}`), it will be included in the JSON output (for *json*, only if *wrap* is `True`).

***wrap* (only for *json*):** if *wrap* is `True` (default), the JSON output is an object of the form `{"count": 2, "results": [...]}`. if `meta` is not `None`, a third property `meta` is added to the wrapping object, with this value.

CHAPTER 3

Contributors

`dataset` is written and maintained by Friedrich Lindenberg, Gregor Aisch and Stefan Wehrmeyer. Its code is largely based on the preceding libraries sqlaload and datafreeze. And of course, we're standing on the shoulders of giants.

Our cute little naked mole rat was drawn by Johannes Koch.

Index