dataset

Release 1.6.2

unknown

Jul 12, 2023

CONTENTS

Although managing data in relational databases 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 that 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

ONE

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

TWO

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 multiple at once
winners = table.find(id=[1, 3, 7])

# Find by comparison operator
elderly_users = table.find(age={'>=': 70})
possible_customers = table.find(age={'between': [21, 80]})
```

(continues on next page)

(continued from previous page)

```
# Use the underlying SQLAlchemy directly
elderly_users = table.find(table.table.columns.age >= 70)
```

See *Advanced filters* for details on complex filters.

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 accessed 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)
```

2.2.7 Limitations of dataset

The goal of `dataset` is to make basic database operations simpler, by expressing some relatively basic operations in a Pythonic way. The downside of this approach is that as your application grows more complex, you may begin to need access to more advanced operations and be forced to switch to using SQLAlchemy proper, without the dataset layer (instead, you may want to play with SQLAlchemy's ORM).

When that moment comes, take the hit. SQLAlchemy is an amazing piece of Python code, and it will provide you with idiomatic access to all of SQL's functions.

Some of the specific aspects of SQL that are not exposed in `dataset`, and are considered out of scope for the project, include:

- Foreign key relationships between tables, and expressing one-to-many and many-to-many relationships in idiomatic Python.
- Python-wrapped `JOIN` queries.

- Creating databases, or managing DBMS software.
- Support for Python 2.x

There's also some functionality that might be cool to support in the future, but that requires significant engineering:

- Async operations
- Database-native `UPSERT` semantics

2.3 API documentation

2.3.1 Connecting

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

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`. 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')
```

One of the main features of *dataset* is to automatically create tables and columns as data is inserted. This behaviour can optionally be disabled via the *ensure_schema* argument. It can also be overridden in a lot of the data manipulation methods using the *ensure* flag.

If you want to run custom SQLite pragmas on database connect, you can add them to on_connect_statements as a set of strings. You can view a full list of PRAGMAs here.

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*, *engine_kwargs=None*, *ensure_schema=True*, *row_type=<class 'collections.OrderedDict'>*, *sqlite_wal_mode=True*, *on_connect_statements=None*)

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*, *primary_increment=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. You can disable auto-increment behaviour for numeric primary keys by setting *primary_increment* to *False*.

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*, *primary_increment=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.

property 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*, *primary_increment=None*, *auto_create=False*)

Represents a table in a database and exposes common operations.

__iter__()

Return all rows of the table as simple dictionaries.

Allows for iterating over all rows in the table without explicitly calling *find()*.

```
for row in table:
    print(row)
```

__len__()

Return the number of rows in the table.

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'])
```

You can also submit filters based on criteria other than equality, see *Advanced filters* for details.

To run more complex queries with JOINs, or to perform GROUP BY-style aggregation, you can also use *db.query()* to run raw SQL queries instead.

property 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*, ***kwargs*)

Create a new column `name` of a specified type.

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

type corresponds to an SQLAlchemy type as described by *dataset.db.Types*. Additional keyword arguments are passed to the constructor of *Column*, so that default values, and options like *nullable* and *unique* can be set.

```
table.create_column('key', unique=True, nullable=False)
table.create_column('food', default='banana')
```

create_column_by_example(*name*, *value*)

Explicitly create a new column `name` with a type that is appropriate to store the given example `value`. The type is guessed in the same way as for the insert method with `ensure=True`.

```
table.create_column_by_example('length', 4.2)
```

If a column of the same name already exists, no action is taken, even if it is not of the type we would have created.

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'])
```

You can also submit filters based on criteria other than equality, see *Advanced filters* for details.

To run more complex queries with JOINs, or to perform GROUP BY-style aggregation, you can also use *db.query()* to run raw SQL queries 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')
```

has_column(*column*)

Check if a column with the given name exists on this table.

has_index(*columns*)

Check if an index exists to cover the given `columns`.

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 no change is made.

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()*.

update_many(*rows*, *keys*, *chunk_size=1000*, *ensure=None*, *types=None*)

Update many rows in the table at a time.

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

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

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'])
```

upsert_many(*rows*, *keys*, *chunk_size=1000*, *ensure=None*, *types=None*)

Sorts multiple input rows into upserts and inserts. Inserts are passed to insert and upserts are updated.

See *upsert()* and *insert_many()*.

2.3.5 Data Export

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

2.4 Advanced filters

`dataset` provides two methods for running queries: *table.find()* and *db.query()*. The table find helper method provides limited, but simple filtering options:

```
results = table.find(column={operator: value})
# e.g.:
results = table.find(name={'like': '%mole rat%'})
```

A special form is using keyword searches on specific columns:

```
results = table.find(value=5)
# equal to:
results = table.find(value={'=': 5})

# Lists, tuples and sets are turned into `IN` queries:
results = table.find(category=('foo', 'bar'))
# equal to:
results = table.find(value={'in': ('foo', 'bar')})
```

The following comparison operators are supported:

Operator	Description
gt, >	Greater than
lt, <	Less than
gte, >=	Greater or equal
lte, <=	Less or equal
!=, <>, not	Not equal to a single value
in	Value is in the given sequence
notin	Value is not in the given sequence
like, ilike	Text search, ILIKE is case-insensitive. Use % as a wildcard
notlike	Like text search, except check if pattern does not exist
between, ..	Value is between two values in the given tuple
startswith	String starts with
endswith	String ends with

Querying for a specific value on a column that does not exist on the table will return no results.

You can also pass additional SQLAlchemy clauses into the *table.find()* method by falling back onto the SQLAlchemy core objects wrapped by *dataset*:

```
# Get the column `city` from the dataset table:
column = table.table.columns.city
# Define a SQLAlchemy clause:
clause = column.ilike('amsterda%')
# Query using the clause:
results = table.find(clause)
```

This can also be used to define combined OR clauses if needed (e.g. *city = 'Bla' OR country = 'Foo'*).

2.4.1 Queries using raw SQL

To run more complex queries with JOINs, or to perform GROUP BY-style aggregation, you can also use `db.query()` to run raw SQL queries instead. This also supports parameterisation to avoid SQL injections.

Finally, you should consider falling back to SQLAlchemy core to construct queries if you are looking for a programmatic, composable method of generating SQL in Python.

CHAPTER

THREE

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