datalite3

Release vv1.0.1

Andrea F. Daniele

Mar 21, 2022

CONTENTS:

1 Documentation 3

2 Indices and tables 11

Python Module Index 13

Index 15

Datalite is a simple to use Python library that can bind a dataclass to an sqlite3 database.

CHAPTER ONE

DOCUMENTATION

1.1 Getting Started

Welcome to the documentation of `datalite3`. `datalite3` provides a simple, intuitive way to bind dataclasses to sqlite3 databases. In its current version, it provides implicit support for conversion between `int`, `float`, `str`, `bytes` classes and their `sqlite3` counterparts, default values, basic schema migration and fetching functions.

1.1.1 Installation

Simply write:

```
pip install datalite3
```

In the shell. And then, whenever you want to use it in Python, you can use:

```
import datalite3
```

1.2 Basic Decorator Operations

1.2.1 Creating a datalite class

A datalite class is a special dataclass. It is created by using a decorator `@datalite3.datalite`, members of this class are, from Python's perspective, just normal classes. However, they have additional methods and attributes. `@datalite` decorator needs a database to be provided. This is the database the table for the dataclass will be created in.

```
from datalite3 import datalite

@datalite('mydb.db')
@dataclass
class Student:
    student_id: int = 1
    student_name: str = "John Smith"
    student_gpa: float = 3.9
```

Here, `datalite` will create a table called `student` in the database file `mydb.db`, this file will include all the fields of the dataclass as columns. Default value of these columns are same as the default value of the dataclass.

1.2.2 Special Methods

Each object initialised from a dataclass decorated with the `@dataclass` decorator automatically gains access to three special methods. It should be noted, due to the nature of the library, extensions such as `mypy` and IDEs such as PyCharm will not be able to see these methods and may raise exceptions.

With this in mind, let us create a new object and run the methods over this objects.

```
new_student = Student(0, "Albert Einstein", 4.0)
```

Creating an Entry

First special method is `.create_entry()` when called on an object of a class decorated with the `@datalite` decorator, this method creates an entry in the table of the bound database of the class, in this case, table named `student` in the `mydb.db`. Therefore, to create the entry of Albert Einstein in the table:

```
new_student.create_entry()
```

Updating an Entry

Second special method is `.update_entry()`. If an object's attribute is changed, to update its record in the database, this method must be called.

```
new_student.student_gpa = 5.0
new_student.update_entry()
```

Deleting an Entry

To delete an entry from the record, the third and last special method, `.remove_entry()` should be used.

```
new_student.remove_entry()
```

1.3 Constraints

One of the most useful features provided by SQLite is the concept of *constraints*. Constraints signal the SQL engine that the values hold in a specific column **MUST** abide by specific constraints, these might be

- Values of this column cannot be `NULL`. (`NOT NULL`)
- Values of this column cannot be repeated. (`UNIQUE`)
- Values of this column must fulfill a condition. (`CHECK`)
- Values of this column can be used to identify a record. (`PRIMARY`)
- Values of this column has a default value. (`DEFAULT`)

Some of these constraints are already implemented in *datalite*. With all of the set, is planned to be implemented in the future.

1.3.1 Default Values

Columns can be given default values. This is done the same way you would give a datafield a default value.

```
@datalite("mydb.db")
@dataclass
class Student:
    student_id: int
    name: str = "Albert Einstein"
```

Therefore, from now on, any `Student` object, whose name is not specified, will have the default name `"Albert Einstein"` and if `.create_entry()` method is called on them, the newly inserted record will, by default, have this value in its corresponding column.

1.3.2 Unique Values

Declaring a field unique is done by a special `TypeVar` called `Unique` this uniqueness check is done in the database level, this introduces has some pros, but also some cons.

Pushing the uniqueness check to the database level introduces a better ability to handle concurrency for applications with large traffic, however, this check only runs when an item is registered, which means no problem will raise in the object creation *even if* another object of the same type with the same value hold in the unique field exists, no error will raise. However, if another *record* with the same value in the unique field is recorded in the bound database, upon the invocation of the `.create_entry()` will raise the `ConstraintFailedError` exception.

Uniqueness constraint is declared thusly:

```
from datalite3 import Unique

@datalite("mydb.db")
@dataclass
class Student:
    student_id: Unique[int]
    name: str = "Albert Einstein"
```

Hinting a field with the `Unique[T]` type variable will introduce two rules:

1. The values in the column of the field in the table that represents the dataclass `Student` in the bound database `mydb.db` cannot be NULL, thus the corresponding field cannot be assigned `None`.
2. These same values **must** be unique for each and every record.

Failure of any of these two rules will result in a `ConstraintFailedError` exception.

1.3.3 Primary Key

In SQL, the PRIMARY KEY constraint uniquely identifies each record in a table. Primary keys must contain UNIQUE values, and cannot contain NULL values. A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

Declaring a field as part of the primary key is done by a special `TypeVar` called `Primary`. For example, in the example of our `Student` class, the primary key can be the field `student_id`, given that a student ID usually uniquely identifies a student within a given institution.

```python
from datalite3 import Primary

@datalite("mydb.db")
@dataclass
class Student:
    student_id: Primary[int]
    name: str = "Albert Einstein"
```

Hinting a field with the `Primary[T]` type variable will introduce two rules:

1. The values in a `Primary` column cannot be NULL, thus the corresponding field cannot be assigned `None`.
2. There cannot be two distinct records in the dataset with the same set of values in the primary key fields.

Failure of any of these two rules will result in a `ConstraintFailedError` exception.

1.4 Fetching Functions

A database is hardly useful if data does not persist between program runs. In `datalite3` one can use `datalite3.fetch` module to fetch data back from the database.

There are different sorts of fetching. One can fetch all the objects of a class using `fetch_all(class_)`, or an object with a specific key using `fetch_from(class_, key)`. There are more functions for plural conditional fetching (`fetch_if`, `fetch_where`) where all objects fitting a condition will be returned, as well as singular conditional fetching that returns the first object that fits a condition (`fetch_equals`).

1.4.1 Pagination

Pagination is a feature that allows a portion of the results to be returned. Since `datalite3` is built to work with databases that may include large amounts of data, many systems using large portions of data also make use of pagination. By building pagination inside the system, we hope to allow easier usage.

- `fetch_where`
- `fetch_if`
- `fetch_all`

Supports pagination, in general, pagination is controlled via two arguments `page` and `element_count`, `page` argument specifies which page to be returned and `element_count` specifies how many elements each page has. When `page` is set to 0, all results are returned irregardless of the value of the `element_count`.

Important: More information regarding the `datalite3.fetch` functions can be found in the API reference.

1.5 Schema Migrations

Datalite provides a module, `datalite3.migrations` that handles schema migrations. When a class definition is modified, `datalite3.migrations.basic_migration` can be called to automatically transfer records to a table fitting the new definitions.

Let us say we have made changes to the fields of a dataclass called `Student` and now, we want these changes to be made to the database. More specifically, we had a field called `studentt_id` and realised this was a typo, we want it to be named into `student_id`, and we want the values that was previously hold in this column to be persistent despite the name change. We can achieve this easily by:

```
datalite3.basic_migration(Student, {'studentt_id': 'student_id'})
```

This will make all the changes, if we had not provided the second argument, the values would be lost.

1.6 API Reference

1.6.1 datalite3 Module

`@datalite3.`**`datalite`**(*db: Union[str, sqlite3.Connection]*, *table_name: Optional[str] = None*, ***, *auto_commit: bool = False*, *type_overload: Optional[Dict[type, datalite3.commons.SQLType]] = None*) → Type[T]

Bind a dataclass to a sqlite3 database. This adds new methods to the class, such as *create_entry()*, *remove_entry()* and *update_entry()*.

Parameters

- **`db`** – Path of the database to be binded.
- **`table_name`** – Optional name for the table. The name of the class will be used by default.
- **`auto_commit`** – Enable auto-commit.
- **`type_overload`** – Type overload dictionary.

Returns The new dataclass.

1.6.2 datalite3.constraints module

datalite3.constraints module introduces constraint types that can be used to hint field variables, that can be used to signal datalite decorator constraints in the database.

`exception` `datalite3.constraints.`**`ConstraintFailedError`**

Bases: `Exception`

This exception is raised when a Constraint fails.

`datalite3.constraints.`**`Unique`**

Dataclass fields hinted with this type signals datalite that the bound column of this field in the table is part of the PRIMARY KEY.

alias of Union[T, Tuple[T]]

1.6.3 datalite3.fetch module

`datalite3.fetch.`**`fetch_all`**(*class_: type*, *page: int = 0*, *element_count: int = 10*) → tuple

Fetchall the records in the bound database.

Parameters

- **`class`** – Class of the records.
- **`page`** – Which page to retrieve, default all. (0 means closed).
- **`element_count`** – Element count in each page.

Returns All the records of type **class_** in the bound database as a tuple.

`datalite3.fetch.`**`fetch_equals`**(*class_: type*, *field: str*, *value: Any*) → Any

Fetch a **class_** type variable from its bound db.

Parameters

- **`class`** – Class to fetch.
- **`field`** – Field to check for, by default, object id.
- **`value`** – Value of the field to check for.

Returns The object whose data is taken from the database.

`datalite3.fetch.`**`fetch_from`**(*class_: type*, *key: Union[None, int, float, str, bytes, Tuple[Union[None, int, float, str, bytes]]]*) → Any

Fetch a **class_** type variable from its bound dv.

Parameters

- **`class`** – Class to fetch from.
- **`key`** – Unique key of the object.

Returns The fetched object.

`datalite3.fetch.`**`fetch_if`**(*class_: type*, *condition: str*, *page: int = 0*, *element_count: int = 10*) → tuple

Fetch all **class_** type variables from the bound db, provided they fit the given condition

Parameters

- **`class`** – Class type to fetch.
- **`condition`** – Condition to check for.
- **`page`** – Which page to retrieve, default all. (0 means closed).
- **`element_count`** – Element count in each page.

Returns A tuple of records that fit the given condition of given type **class_**.

`datalite3.fetch.`**`fetch_where`**(*class_: type*, *field: str*, *value: Any*, *page: int = 0*, *element_count: int = 10*) → tuple

Fetch all **class_** type variables from the bound db, provided that the field of the records fit the given value.

Parameters

- **`class`** – Class of the records.
- **`field`** – Field to check.
- **`value`** – Value to check for.
- **`page`** – Which page to retrieve, default all. (0 means closed).

- **`element_count`** – Element count in each page.

Returns A tuple of the records.

`datalite3.fetch.`**`is_fetchable`**(*class_: type, key: Union[None, int, float, str, bytes, Tuple[Union[None, int, float, str, bytes]]]*) → bool

Check if a record is fetchable given its key and **class_** type.

Parameters

- **`class`** – Class type of the object.
- **`key`** – Unique key of the object.

Returns If the object is fetchable.

1.6.4 datalite3.mass_actions module

This module includes functions to insert multiple records to a bound database at one time, with one time open and closing of the database file.

`exception` `datalite3.mass_actions.`**`HeterogeneousCollectionError`**

Bases: `Exception`

:raise [if the passed collection is not homogeneous.] ie: If a List or Tuple has elements of multiple types.

`datalite3.mass_actions.`**`create_many`**(*objects: Union[List[T], Tuple[T]]*) → None

Insert many records corresponding to objects in a tuple or a list.

Parameters **`objects`** – A tuple or a list of objects decorated with datalite.

Returns None.

1.6.5 datalite3.migrations module

Migrations module deals with migrating data when the object definitions change. This functions deal with Schema Migrations.

`datalite3.migrations.`**`basic_migrate`**(*class_: Type[dataclasses.dataclass], column_transfer: dict = None*) → None

Given a class, compare its previous table, delete the fields that no longer exist, create new columns for new fields. If the column_flow parameter is given, migrate elements from previous column to the new ones.

Parameters

- **`class`** – Datalite class to migrate.
- **`column_transfer`** – A dictionary showing which columns will be copied to new ones.

Returns None.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

datalite3.constraints, 7
datalite3.fetch, 8
datalite3.mass_actions, 9
datalite3.migrations, 9

INDEX

B

basic_migrate() (*in module datalite3.migrations*), 9

C

ConstraintFailedError, 7
create_many() (*in module datalite3.mass_actions*), 9

D

datalite() (*in module datalite3*), 7
datalite3.constraints
  module, 7
datalite3.fetch
  module, 8
datalite3.mass_actions
  module, 9
datalite3.migrations
  module, 9

F

fetch_all() (*in module datalite3.fetch*), 8
fetch_equals() (*in module datalite3.fetch*), 8
fetch_from() (*in module datalite3.fetch*), 8
fetch_if() (*in module datalite3.fetch*), 8
fetch_where() (*in module datalite3.fetch*), 8

H

HeterogeneousCollectionError, 9

I

is_fetchable() (*in module datalite3.fetch*), 9

M

module
  datalite3.constraints, 7
  datalite3.fetch, 8
  datalite3.mass_actions, 9
  datalite3.migrations, 9

U

Unique (*in module datalite3.constraints*), 7