wurm

Release 0.1.0

Jasmijn Wellner

May 26, 2021

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Installation

wurm is distributed on PyPI as a universal wheel and is available on Linux/macOS and Windows and supports Python 3.7+.

```
$ pip install wurm
```

1.2 First steps

To get started with Wurm, let's first create a table:

```
from dataclasses import dataclass
from wurm import Table, Unique

@dataclass
class NamedPoint(Table):
    x: int
    y: int
    name: Unique[str]
```

Alright, so this tells Wurm what a `NamedPoint` is, that it has two regular fields named `x` and `y` which should both be integers, and a field named `name` which has a *Unique* constraint and should be a string.

To anything interesting with it, we should connect to a database, though. SQL databases are usually stored in a file, but if we pass `':memory:'` as the filename, sqlite creates a temporary database in RAM, which is useful for quick tests and trying things out.

```
from wurm import setup_connection
import sqlite3

setup_connection(sqlite3.connect(':memory:'))
```

Now, we can create objects, insert them in the database, and try some simple queries:

```
basecamp = NamedPoint(x=1, y=2, name='Basecamp')

print(basecamp)

```

(continues on next page)

(continued from previous page)

```
basecamp.insert()

print(basecamp.rowid)

NamedPoint(x=10, y=-7, name='Goal').insert()

print(list(NamedPoint))
print(NamedPoint.query(x=10).one())
```

Which produces the following output:

```
NamedPoint(x=1, y=2, name='Basecamp')
1
[NamedPoint(x=1, y=2, name='Basecamp'), NamedPoint(x=10, y=-7, name='Goal')]
NamedPoint(x=10, y=-7, name='Goal')
```

TODO: explain commit, delete, queries /w comparators, show errors from Unique constraint violations and other errors

CHAPTER

TWO

API REFERENCE

2.1 Connecting to a database

wurm.**setup_connection**(*conn*)

Call this once in each OS thread with a `sqlite3.Connection`, before accessing the database via wurm.

This records the connection and ensures all tables are created.

2.2 Defining tables

class wurm.tables.**BaseTable**(**args*, ***kwargs*)

Baseclass for your own tables. Tables must be dataclasses.

Do not use directly, subclass *wurm.Table* or *wurm.WithoutRowid* instead.

Use the keyword argument *name* in the class definition to set the table name:

```
@dataclass
class MyTable(Table, name='mytable'):
    ...
```

If not given, wurm uses the class name to automatically derive a suitable table name.

Use the keyword argument *abstract* in the class definition to add fields or methods that you want to share between several tables:

```
@dataclass
class HasOwner(Table, abstract=True):
    owner: str
    def display_owner(self):
        return self.owner.capitalize()
```

The above will not create a new table, but subclasses of `HasOwner` will have a field called `owner` and a method called `display_owner`.

Models are context managers, so one could write:

```
with MyTable.query(field=value) as obj:
    obj.field = other_value
```

This automatically calls *commit()* on `obj` as long as no exception is raised inside the `with`-block, in which case `obj` is left in a dirty state.

classmethod query(***kwargs*)
Create a query object.

The names of keywords passed should be *rowid* or any of the fields defined on the table.

The values can either be Python values matching the types of the relevant fields, or the same wrapped in one of *lt()*, *gt()*, *le()*, *ge()*, *eq()* or *ne()*. When unwrapped, the behavior matches that of values wrapped in *eq()*.

Merely creating a query does not access the database.

Returns A query for this table.

Return type *Query*

classmethod __len__()
The total number of rows in this table. A shortcut for `len(table.query())`.

Note: This method accesses the connected database.

classmethod __iter__()
Iterate over all the objects in the table. A shortcut for `iter(table.query())`

Note: This method accesses the connected database.

commit()
Commits any changes to the object to the database.

Note: This method accesses the connected database.

delete()
Deletes this object from the database.

Note: This method accesses the connected database.

Raises **ValueError** – if called twice on the same instance, or called on a fresh instance that has not been inserted yet.

insert()
Insert a new object into the database.

Note: This method accesses the connected database.

class wurm.**Table**
Baseclass for regular rowid tables. See *BaseTable* for methods available on subclasses.

class wurm.**WithoutRowid**
Baseclass for `WITHOUT ROWID` tables. You need to add an explicit primary key using *Primary* for these kinds of tables. See *BaseTable* for methods available on subclasses.

2.2.1 Annotations

Tables are defined using **PEP 526** type annotations, where the type for each column has to be one of the following:

- One of the basic supported types (currently `str`, `bytes`, `int`, `float`, `bool`, `datetime.date`, `datetime.time`, `datetime.datetime` and `pathlib.Path`).
- A type registered with *wurm.register_type()*.
- A previously defined *wurm.Table* or *wurm.WithoutRowid* subclass.
- `wurm.Primary[T]`, `wurm.Index[T]` or `wurm.Unique[T]`, where *T* is one of the types mentioned above.

wurm.Primary
Using `Primary[T]` as a type annotation in a table definition is equivalent to using *T*, except that the column will be part of the primary key. If multiple fields on a single table definition are annotated in this way, their columns form a composite primary key together.

If you attempt change the database in a way that would cause two rows to share a primary key, the operation is rolled back, and a *WurmError* is raised.

wurm.Index
Using `Index[T]` as a type annotation in a table definition is equivalent to using *T*, except that a (non-`UNIQUE`) index is created for the field.

wurm.Unique
Using `Unique[T]` as a type annotation in a table definition is equivalent to using *T*, except that a `UNIQUE` index is created for the field. Note that SQL considers `None` values to be different from other `None` values for this purpose.

If you attempt to call *insert()* or *commit()* in a way that would violate such a constraint, the operation is rolled back, and a *WurmError* is raised.

wurm.register_type(*python_type*, *sql_type*, *, *encode*, *decode*)
Registers a type for use in model fields.

For example:

```
class Foo:
    def __repr__(self):
        ...
    @classmethod
    def from_string(cls, string):
        ...
    ...

register_type(Foo, str, encode=repr, decode=Foo.from_string)
```

Parameters

- **python_type** (*type*) – The type to register
- **sql_type** – The stored type, one of `int`, `str`, `float`, `bytes`, a dict from `str` to the above primitive types, or a tuple of the above primitive types. The latter two options is for types that should be mapped to multiple columns.
- **encode** (*python_type -> sql_type*) – The function to prepare to store a value in the database. Should return a tuple if the type is mapped to multiple columns.
- **decode** (*sql_type -> python_type*) – The function to interpret the stored value. Should take multiple arguments in the case types mapped to multiple columns.

wurm.**register_dataclass**(*dclass*)

Registers a dataclass for use in model fields.

This is a convenience function that can optionally be used as a decorator. Given:

```
@dataclasses.dataclass
class Color:
    r: float
    g: float
    b: float
```

then the following:

```
register_dataclass(Color)
```

is equivalent to:

```
register_type(Color, dict(r=float, g=float, b=float),
    encode=dataclasses.astuple, decode=Color)
```

In either case, the model:

```
class MyTable(Table):
    color: Color
```

will have the fields `color_r`, `color_g` and `color_b`, which will transparently be converted to and from `Color` objects.

Parameters **dclass** – The dataclass to register

Returns The registered dataclass

2.3 Queries

2.3.1 Query objects

Most advanced queries will be done through `Query` objects, that can be created either explicitly through their constructor, or by calling `Table.query()`.

class wurm.**Query**(*table: Type[wurm.queries.T]*, *filters: Dict[str, Any]*)

Represents one or more queries on a specified table.

`Query(table, filters)` is equivalent to `table.query(**filters)`

__iter__() → Iterator[wurm.queries.T]

Iterate over the results of this query.

Note: This method accesses the connected database.

Equivalent to *select_with_limit()* without specifying *limit*.

__len__() → int

Returns the number of rows matching this query.

Note: This method accesses the connected database.

Returns number of matches

Return type int

delete() → None
Delete the objects matching this query.

Warning: Calling this on an empty query deletes all rows of the relevant table in the database

Note: This method accesses the connected database.

Returns the number of rows deleted

Return type int

first() → wurm.queries.T
Return the first result of this query.

Note: This method accesses the connected database.

Raises *WurmError* – if this query returns zero results

one() → wurm.queries.T
Return the only result of this query.

Note: This method accesses the connected database.

Raises *WurmError* – if this query returns zero results or more than one

select_with_limit(*limit: Optional[int] = None*) → Iterator[wurm.queries.T]
Create an iterator over the results of this query.

Note: This method accesses the connected database.

Parameters **limit** (*int or None*) – The number of results to limit this query to.

Returns an iterator over the objects matching this query.

2.3.2 Comparators

wurm.**lt**(*value*)
wurm.**le**(*value*)
wurm.**eq**(*value*)
wurm.**ne**(*value*)
wurm.**ge**(*value*)
wurm.**gt**(*value*)

Used to wrap values in queries. These functions correspond to the special names for the Python comparison operators.

The expression

```
MyTable.query(a=le(1), b=gt(2), c=3, d=ne(4))
```

is roughly equivalent to

```
SELECT * FROM MyTable WHERE a <= 1 AND b > 2 AND c = 3 AND d != 4
```

Replacing `c=3` with `c=eq(3)` is optional.

2.3.3 Exceptions

exception wurm.**WurmError**

General error for a database operation failing.

Its `__cause__` attribute refers to the relevant `sqlite3.Error` when that exists.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX