
itemdb
Release 1.1.2

Apr 20, 2023

Contents:

1	Installing itemdb	3
2	Guide	5
2.1	Introduction	5
2.2	Opening a database	5
2.3	Creating tables and indices	6
2.4	Add some items	6
2.5	Make some queries	7
2.6	Avoiding SQL injection	7
2.7	Transactions	7
2.8	Database maintenance	8
2.9	Going Async	8
3	Reference	9
3.1	The ItemDB class	9
3.2	The AsyncItemDB class	12
3.3	The asyncify function	13
4	Indices and tables	15
	Python Module Index	17
	Index	19

The itemdb library allows you to store and retrieve Python dicts in a database on the local filesystem, in an easy, fast, and reliable way.

Based on the rock-solid and ACID compliant SQLite, but with easy and explicit transactions using a `with` statement. It provides a simple object-based API, with the flexibility to store (JSON-compatible) items with arbitrary fields, and add indices when needed.

CHAPTER 1

Installing itemdb

The itemdb library is pure Python and has zero dependencies.

```
pip install itemdb
```


2.1 Introduction

The `itemdb` library allows you to store and retrieve Python dicts in a database on the local filesystem, in an easy, fast, and reliable way.

To be more precise: it is an **ACID compliant** transactional database for storage and retrieval of JSON-compatible dict items. That sounds very technical; let's break it down:

- ACID means it has desirable database features of atomicity, consistency, isolation and durability. We'll get back to these when we talk about transactions.
- The `itemdb` API focuses on making transactions easy and explicit.
- JSON is used to serialize the dict items, so the values in the dicts are limited to: `None`, `bool`, `int`, `float`, `str`, `list`, `dict`.

In practice, `itemdb` uses the rock-solid **SQLite**, and provides an object-based API that requires no knowledge of SQL.

You can use `itemdb` in a wide variety of applications. This includes web-servers, though once your traffic scales up, you may want to consider something like PostgreSQL or perhaps a hosted db.

2.2 Opening a database

In `itemdb` (like SQLite) each database is represented as a file. One can also use `":memory:"` to create an in-memory database for testing/demo purposes.

```
db = ItemDB(filename)
db = ItemDB(":memory:")
```

2.3 Creating tables and indices

Each database consists of tables, and the tables contain the items. A “table” is what is also called “table” in SQL databases, a “collection” in MongoDB, and an “object store” in IndexedDB.

You can create a table using `ensure_table()`. It is safe to call this before every time that you use the database, because it returns fast if the table already exist:

```
db.ensure_table("some_table_name")
```

In the same call we can also specify indices. Indices represent fields in the items that are indexed, so that they can be used to retrieve items fast, using `select()`, `count()` and `delete()`.

Indices can also be prefixed with a “!”, marking the field as mandatory and unique, making it possible to identify items.

```
db.ensure_table("persons", "!name", "age")
```

We can now `select()` items based on the name and age fields, and no two items can have the same value for name.

Note: No new fields can be marked unique once the table has been created.

Note: In the examples below we mark the “name” field as unique, but strictly speaking this is wrong, because different persons can have the same name. Another form of ID would be more appropriate in real use-cases.

2.4 Add some items

An “item” is what is called a “row” in SQL databases, a “document” in MongoDB, and an “object” in IndexedDB. Let’s add some to our table!

```
with db:
    db.put_one("persons", name="Jane", age=22)
    db.put_one("persons", name="John", age=18, fav_number=7)
    db.put("persons", {"name": "Guido"}, {"name": "Anne", "age": 42})
```

You can see how we use `with db` here. This is because `itemdb` requires using a transaction when making changes to the database. Everything inside the `with` statement is a single transaction. More on that later.

You can also see that with `put_one()` we can use keyword arguments to specify fields, while with `put()` we can specify multiple items, each items a dict.

The dictionary can contain as many fields as you want, including sub-dicts and lists. Although the `age` field is indexed, it is not mandatory (you can select items with missing age using `db.select("persons", "age is NULL")`).

Since the `name` field is unique, if we put an item with an existing name, it will simply update it:

```
# John had his birthday and changed his favourite number
with db:
    db.put_one("persons", name="John", age=19, fav_number=8)
```

2.5 Make some queries

Use e.g. `count()`, `select()` to query the database:

```
>>> db.count_all("persons")
4

>>> db.select("persons", "age > ?", 20)
[{'name': 'Jane', 'age': 22}, {'name': 'Anne', 'age': 42}]

>>> select_name = "John"
>>> db.select_one("persons", "name = ?", select_name)
{'name': 'John', 'age': 19, 'fav_number': 8}
```

2.6 Avoiding SQL injection

SQL injection is a technique by which a potential hacker could access your database to get access or destroy data. The common path for SQL injection is to write SQL code in an end-user input field.

For example, imagine a website that sells paintings, which may have a field for the minimum size. You'll want to use the given size in a query (e.g. a `db.select()`) so you can show the user all paintings that qualify. Now imagine that an attacker writes SQL code in that input field. If the input is not *sanitized*, your db is compromised!

This is the reason for the `?` notation used throughout these docs - the actual arguments are passed to SQLite in a safe way. It's a good habit to *always* provide query arguments this way.

2.7 Transactions

Transactions are an important concept in databases. In ACID databases (like itemdb) it has a number of features:

- A transaction is atomic (either the whole transaction is applied, or the whole transaction is not applied)
- A transaction is applied in isolation, even when multiple processes are interacting with the database at the same time. This means that when a transaction is in progress, another process/thread that wants to apply a transaction that “intersects” with the ongoing operation, it will wait. (This even works for multiple Docker containers operating on the same SQLite database.)
- The remaining elements of ACID (consistency and durability) mean that the database always remains in a healthy state. Even on a power outage or if the system crashes halfway a transaction.

In itemdb, transactions are easy, using a context manager. Let's have a look at some examples:

```
# Increasing a value is recommended to do in a transaction.
with db:
    player = db.select("players", "name == ?", player_name)
    player["position"] += 2
    db.put("players", player)

# The below has no effect: the transaction fails and is rolled back
with db:
    db.put_one("persons", name="John", age=21, fav_number=8)
    raise RuntimeError()
```

2.8 Database maintenance

Sometimes, you may want to add unique keys to a table or remove existing indices. This is possible by copying the items to a new table and then replacing the new table with the old. By doing this inside a transaction, it can be done safely:

```
with db:
    db.ensure_table("persons2", "!id", "name", "age")
    for i, person in enumerate(db.select_all("persons")):
        # Make sure each person has an id, e.g.:
        person["id"] = i
        db.put("persons2", person)
    db.delete_table("persons")
    db.rename_table("persons2", "persons")
```

At the time of writing, itemdb does not provide an API for `backups` or `vacuuming`, but it's just SQLite under the hood, so you can use the common methods.

2.9 Going Async

The API of `ItemDB` is synchronous. It operates with the filesystem, so it can benefit from async use a lot.

There are two ways to make your code async. The first is by using the `AsyncItemDB` class. It has the exact same API as `ItemDB`, but all its methods are async. Note that you must also use `async with`.

The second approach is to asyncify a synchronous function. The idea of this approach is to do all itemdb operations inside a function and then wrap that function if you want to use it in an async environment. Consider the following example of a web server:

```
@itemdb.asyncify
def push_items(filename, items):
    db = ItemDB(filename)
    db.ensure_table("my_table", "!id", "mtime")

    with db:
        ...
        db.put("my_table", items)

async def my_request_handler(request):
    ...
    # Because we decorated the function with asyncify,
    # we can now await it, while the db interaction
    # occurs in a separate thread.
    await push_items(filename, items)
    ...
```

Of the two mentioned approaches, the `asyncify`-approach is slightly more efficient, because it makes use of a thread pool, and only switches to a thread for the duration of the function you've asyncified. However, using `AsyncItemDB` probably makes your code easier to read and maintain, which is probably worth more.

3.1 The ItemDB class

class `itemdb.ItemDB` (*filename*)

A transactional database for storage and retrieval of dict items.

Parameters `filename` (*str*) – The file to open. Use “:memory:” for an in-memory db.

The items in the database can be any JSON serializable dictionary. Indices can be defined for specific fields to enable fast selection of items based on these fields. Indices can be marked as unique to make a field mandatory and *identify* items based on that field.

Transactions are done by using the `with` statement, and are mandatory for all operations that write to the database.

mtime

The time that the database file was last modified, as a Unix timestamp.

Is -1 if the file did not exist, or if the filename is not represented on the filesystem.

close ()

Close the database connection.

This will be automatically called when the instance is deleted. But since it can be held e.g. in a traceback, consider using `with closing(db) :`

get_table_names ()

Return a (sorted) list of table names present in the database.

get_indices (*table_name*)

Get a set of index names for the given table.

Parameters `table_name` (*str*) – The name of the table to get the indices for. *To avoid SQL injection, this arg should not be based on unsafe data.*

Names prefixed with “!” represent fields that are required and unique. Raises `KeyError` if the table does not exist.

ensure_table (*table_name*, **indices*)

Ensure that the given table exists and has the given indices.

Parameters

- **table_name** (*str*) – The name of the table to make sure exists. *To avoid SQL injection, this arg should not be based on unsafe data.*
- **indices** (*varargs*) – A sequence of strings, representing index names. Fields that are indexed can be queried with e.g. `select ()`. *To avoid SQL injection, this arg should not be based on unsafe data.*

If an index name is prefixed with “!”, it indicates a field that is mandatory and unique. Note that new unique indices cannot be added when the table already exist.

This method returns as quickly as possible when the table already exists and has the appropriate indices. Returns the ItemDB object, so calls to this method can be stacked.

Although this call may modify the database, one does not need to call this in a transaction.

delete_table (*table_name*)

Delete the table with the given name.

Parameters **table_name** (*str*) – The name of the table to delete. *To avoid SQL injection, this arg should not be based on unsafe data.*

Be aware that this deletes the whole table, including all of its items.

This method must be called within a transaction. Can raise `KeyError` if an invalid table is given, or `IOError` if not used within a transaction

rename_table (*table_name*, *new_table_name*)

Rename a table.

Parameters

- **table_name** (*str*) – The current name of the table. *To avoid SQL injection, this arg should not be based on unsafe data.*
- **new_table_name** (*str*) – The new name. *To avoid SQL injection, this arg should not be based on unsafe data.*

This method must be called within a transaction. Can raise `KeyError` if an invalid table is given, or `IOError` if not used within a transaction

count_all (*table_name*)

Get the total number of items in the given table.

count (*table_name*, *query*, **save_args*)

Get the number of items in the given table that match the given query.

Parameters

- **table_name** (*str*) – The name of the table to count items in. *To avoid SQL injection, this arg should not be based on unsafe data.*
- **query** (*str*) – The query to select items on. *To avoid SQL injection, this arg should not be based on unsafe data; use save_args for end-user input.*
- **save_args** (*varargs*) – The values to select items on.

Examples:

```
# Count the persons older than 20
db.count("persons", "age > ?", 20)
# Count the persons older than a given value
db.count("persons", "age > ?", min_age)
# Use AND and OR for more precise queries
db.count("persons", "age > ? AND age < ?", min_age, max_age)
```

See `select()` for details on queries.

Can raise `KeyError` if an invalid table is given, `IndexError` if an invalid field is used in the query, or `sqlite3.OperationalError` for an invalid query.

select_all (*table_name*)

Get all items in the given table. See `select()` for details.

select (*table_name*, *query*, **save_args*)

Get the items in the given table that match the given query.

Parameters

- **table_name** (*str*) – The name of the table to select items in. *To avoid SQL injection, this arg should not be based on unsafe data.*
- **query** (*str*) – The query to select items on. *To avoid SQL injection, this arg should not be based on unsafe data; use save_args for end-user input.*
- **save_args** (*varargs*) – The values to select items on.

The query follows SQLite syntax and can only include indexed fields. If needed, use `ensure_table()` to add indices. The query is always fast (which is why this method is called ‘select’, and not ‘search’).

Examples:

```
# Select the persons older than 20
db.select("persons", "age > ?", 20)
# Select the persons older than a given age
db.select("persons", "age > ?", min_age)
# Use AND and OR for more precise queries
db.select("persons", "age > ? AND age < ?", min_age, max_age)
```

There is no method to filter items bases on non-indexed fields, because this is easy using a list comprehension, e.g.:

```
items = db.select_all("persons")
items = [i for i in items if i["age"] > 20]
```

Can raise `KeyError` if an invalid table is given, `IndexError` if an invalid field is used in the query, or `sqlite3.OperationalError` for an invalid query.

select_one (*table_name*, *query*, **args*)

Get the first item in the given table that match the given query.

Parameters

- **table_name** (*str*) – The name of the table to select an item in. *To avoid SQL injection, this arg should not be based on unsafe data.*
- **query** (*str*) – The query to select the item on. *To avoid SQL injection, this arg should not be based on unsafe data; use save_args for end-user input.*
- **save_args** (*varargs*) – The values to select the item on.

Returns None if there was no match. See `select()` for details.

put (*table_name*, **items*)

Put one or more items into the given table.

Parameters

- **table_name** (*str*) – The name of the table to put the item(s) in. *To avoid SQL injection, this arg should not be based on unsafe data.*
- **items** (*varargs*) – The dicts to add. Keys that match an index can later be used for fast querying.

This method must be called within a transaction. Can raise `KeyError` if an invalid table is given, `IOError` if not used within a transaction, `TypeError` if an item is not a (JSON serializable) dict, or `IndexError` if an item does not have a required field.

put_one (*table_name*, ***item*)

Put an item into the given table using kwargs.

Parameters

- **table_name** (*str*) – The name of the table to put the item(s) in. *To avoid SQL injection, this arg should not be based on unsafe data.*
- **item** (*kwargs*) – The dict to add. Keys that match an index can later be used for fast querying.

This method must be called within a transaction.

delete (*table_name*, *query*, **save_args*)

Delete items from the given table.

Parameters

- **table_name** (*str*) – The name of the table to delete items from. *To avoid SQL injection, this arg should not be based on unsafe data.*
- **query** (*str*) – The query to select the items to delete. *To avoid SQL injection, this arg should not be based on unsafe data; use save_args for end-user input.*
- **save_args** (*varargs*) – The values to select the item on.

Examples:

```
# Delete the persons older than 20
db.delete("persons", "age > ?", 20)
# Delete the persons older than a given age
db.delete("persons", "age > ?", min_age)
# Use AND and OR for more precise queries
db.delete("persons", "age > ? AND age < ?", min_age, max_age)
```

See `select()` for details on queries.

This method must be called within a transaction. Can raise `KeyError` if an invalid table is given, `IOError` if not used within a transaction, `IndexError` if an invalid field is used in the query, or `sqlite3.OperationalError` for an invalid query.

3.2 The AsyncItemDB class

class `itemdb.AsyncItemDB`

An async version of `ItemDB`. The API is exactly the same, except that all methods are async, and one must use

async with instead of the normal *with*.

3.3 The `asyncify` function

`itemdb.asyncify(func)`

Wrap a normal function into an awaitable co-routine. Can be used as a decorator.

The original function will be executed in a separate thread. This allows async code to execute io-bound code (like querying a sqlite database) without stalling.

Note that the code in `func` must be thread-safe. It's probably best to isolate the io-bound parts of your code and only wrap these.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

i

itertools, ??

A

`asyncify()` (*in module itemdb*), 13
`AsyncItemDB` (*class in itemdb*), 12

C

`close()` (*itemdb.ItemDB method*), 9
`count()` (*itemdb.ItemDB method*), 10
`count_all()` (*itemdb.ItemDB method*), 10

D

`delete()` (*itemdb.ItemDB method*), 12
`delete_table()` (*itemdb.ItemDB method*), 10

E

`ensure_table()` (*itemdb.ItemDB method*), 9

G

`get_indices()` (*itemdb.ItemDB method*), 9
`get_table_names()` (*itemdb.ItemDB method*), 9

I

`ItemDB` (*class in itemdb*), 9
`itemdb` (*module*), 1

M

`mtime` (*itemdb.ItemDB attribute*), 9

P

`put()` (*itemdb.ItemDB method*), 12
`put_one()` (*itemdb.ItemDB method*), 12

R

`rename_table()` (*itemdb.ItemDB method*), 10

S

`select()` (*itemdb.ItemDB method*), 11
`select_all()` (*itemdb.ItemDB method*), 11
`select_one()` (*itemdb.ItemDB method*), 11