TinyDB Documentation

Release 4.5.2

Markus Siemens

Sep 23, 2021

Contents

Welcome to TinyDB, your tiny, document oriented database optimized for your happiness :)

```
>>> from tinydb import TinyDB, Query
>>> db = TinyDB('path/to/db.json')
>>> User = Query()
>>> db.insert({'name': 'John', 'age': 22})
>>> db.search(User.name == 'John')
[{'name': 'John', 'age': 22}]
```

CHAPTER 1

User's Guide

1.1 Introduction

Great that you've taken time to check out the TinyDB docs! Before we begin looking at TinyDB itself, let's take some time to see whether you should use TinyDB.

1.1.1 Why Use TinyDB?

- **tiny:** The current source code has 1800 lines of code (with about 40% documentation) and 1600 lines tests.
- **document oriented:** Like MongoDB, you can store any document (represented as `dict`) in TinyDB.
- **optimized for your happiness:** TinyDB is designed to be simple and fun to use by providing a simple and clean API.
- **written in pure Python:** TinyDB neither needs an external server (as e.g. PyMongo) nor any dependencies from PyPI.
- **works on Python 3.5+ and PyPy:** TinyDB works on all modern versions of Python and PyPy.
- **powerfully extensible:** You can easily extend TinyDB by writing new storages or modify the behaviour of storages with Middlewares.
- **100% test coverage:** No explanation needed.

In short: If you need a simple database with a clean API that just works without lots of configuration, TinyDB might be the right choice for you.

1.1.2 Why Not Use TinyDB?

- **You need advanced features like:**
  - access from multiple processes or threads,
  - creating indexes for tables,

  - an HTTP server,
  - managing relationships between tables or similar,
  - ACID guarantees.
- You are really concerned about **performance** and need a high speed database.

To put it plainly: If you need advanced features or high performance, TinyDB is the wrong database for you – consider using databases like SQLite, Buzhug, CodernityDB or MongoDB.

1.2 Getting Started

1.2.1 Installing TinyDB

To install TinyDB from PyPI, run:

```
$ pip install tinydb
```

You can also grab the latest development version from GitHub. After downloading and unpacking it, you can install it using:

```
$ pip install .
```

1.2.2 Basic Usage

Let's cover the basics before going more into detail. We'll start by setting up a TinyDB database:

```
>>> from tinydb import TinyDB, Query
>>> db = TinyDB('db.json')
```

You now have a TinyDB database that stores its data in `db.json`. What about inserting some data? TinyDB expects the data to be Python `dicts`:

```
>>> db.insert({'type': 'apple', 'count': 7})
>>> db.insert({'type': 'peach', 'count': 3})
```

Note: The `insert` method returns the inserted document's ID. Read more about it here: *Using Document IDs*.

Now you can get all documents stored in the database by running:

```
>>> db.all()
[{'count': 7, 'type': 'apple'}, {'count': 3, 'type': 'peach'}]
```

You can also iter over stored documents:

```
>>> for item in db:
>>>     print(item)
{'count': 7, 'type': 'apple'}
{'count': 3, 'type': 'peach'}
```

Of course you'll also want to search for specific documents. Let's try:

```
>>> Fruit = Query()
>>> db.search(Fruit.type == 'peach')
[{'count': 3, 'type': 'peach'}]
>>> db.search(Fruit.count > 5)
[{'count': 7, 'type': 'apple'}]
```

Next we'll update the `count` field of the apples:

```
>>> db.update({'count': 10}, Fruit.type == 'apple')
>>> db.all()
[{'count': 10, 'type': 'apple'}, {'count': 3, 'type': 'peach'}]
```

In the same manner you can also remove documents:

```
>>> db.remove(Fruit.count < 5)
>>> db.all()
[{'count': 10, 'type': 'apple'}]
```

And of course you can throw away all data to start with an empty database:

```
>>> db.truncate()
>>> db.all()
[]
```

Recap

Before we dive deeper, let's recapitulate the basics:

Inserting	
`db.insert(...)`	Insert a document
Getting data	
`db.all()`	Get all documents
`iter(db)`	Iter over all documents
`db.search(query)`	Get a list of documents matching the query
Updating	
`db.update(fields, query)`	Update all documents matching the query to contain `fields`
Removing	
`db.remove(query)`	Remove all documents matching the query
`db.truncate()`	Remove all documents
Querying	
`Query()`	Create a new query object
`Query().field == 2`	Match any document that has a key `field` with value `== 2` (also possible: `!=`, `>`, `>=`, `<`, `<=`)

1.3 Advanced Usage

1.3.1 Remarks on Storage

Before we dive deeper into the usage of TinyDB, we should stop for a moment and discuss how TinyDB stores data.

To convert your data to a format that is writable to disk TinyDB uses the Python JSON module by default. It's great when only simple data types are involved but it cannot handle more complex data types like custom classes. On Python 2 it also converts strings to Unicode strings upon reading (described here).

If that causes problems, you can write *your own storage*, that uses a more powerful (but also slower) library like pickle or PyYAML.

Hint: Opening multiple TinyDB instances on the same data (e.g. with the `JSONStorage`) may result in unexpected behavior due to query caching. See *query_caching* on how to disable the query cache.

1.3.2 Queries

With that out of the way, let's start with TinyDB's rich set of queries. There are two main ways to construct queries. The first one resembles the syntax of popular ORM tools:

```
>>> from tinydb import Query
>>> User = Query()
>>> db.search(User.name == 'John')
```

As you can see, we first create a new Query object and then use it to specify which fields to check. Searching for nested fields is just as easy:

```
>>> db.search(User.birthday.year == 1990)
```

Not all fields can be accessed this way if the field name is not a valid Python identifier. In this case, you can switch to array indexing notation:

```
>>> # This would be invalid Python syntax:
>>> db.search(User.country-code == 'foo')
>>> # Use this instead:
>>> db.search(User['country-code'] == 'foo')
```

The second, traditional way of constructing queries is as follows:

```
>>> from tinydb import where
>>> db.search(where('field') == 'value')
```

Using `where('field')` is a shorthand for the following code:

```
>>> db.search(Query()['field'] == 'value')
```

Accessing nested fields with this syntax can be achieved like this:

```
>>> db.search(where('birthday').year == 1900)
>>> db.search(where('birthday')['year'] == 1900)
```

Advanced queries

In the *Getting Started* you've learned about the basic comparisons (==, <, >, . . .). In addition to these TinyDB supports the following queries:

```
>>> # Existence of a field:
>>> db.search(User.name.exists())
```

```
>>> # Regex:
>>> # Full item has to match the regex:
>>> db.search(User.name.matches('[aZ]*'))
>>> # Case insensitive search for 'John':
>>> import re
>>> db.search(User.name.matches('John', flags=re.IGNORECASE))
>>> # Any part of the item has to match the regex:
>>> db.search(User.name.search('b+'))
```

```
>>> # Custom test:
>>> test_func = lambda s: s == 'John'
>>> db.search(User.name.test(test_func))
```

```
>>> # Custom test with parameters:
>>> def test_func(val, m, n):
>>>     return m <= val <= n
>>> db.search(User.age.test(test_func, 0, 21))
>>> db.search(User.age.test(test_func, 21, 99))
```

Another case is if you have a `dict` where you want to find all documents that match this `dict`. We call this searching for a fragment:

```
>>> db.search(Query().fragment({'foo': True, 'bar': False}))
[{'foo': True, 'bar': False, 'foobar: 'yes!'}]
```

You also can search for documents where a specific field matches the fragment:

```
>>> db.search(Query().field.fragment({'foo': True, 'bar': False}))
[{'field': {'foo': True, 'bar': False, 'foobar: 'yes!'}]
```

When a field contains a list, you also can use the `any` and `all` methods. There are two ways to use them: with lists of values and with nested queries. Let's start with the first one. Assuming we have a user object with a groups list like this:

```
>>> db.insert({'name': 'user1', 'groups': ['user']})
>>> db.insert({'name': 'user2', 'groups': ['admin', 'user']})
>>> db.insert({'name': 'user3', 'groups': ['sudo', 'user']})
```

Now we can use the following queries:

```
>>> # User's groups include at least one value from ['admin', 'sudo']
>>> db.search(User.groups.any(['admin', 'sudo']))
[{'name': 'user2', 'groups': ['admin', 'user']},
 {'name': 'user3', 'groups': ['sudo', 'user']}]
>>>
>>> # User's groups include all values from ['admin', 'user']
>>> db.search(User.groups.all(['admin', 'user']))
[{'name': 'user2', 'groups': ['admin', 'user']}]
```

In some cases you may want to have more complex `any`/`all` queries. This is where nested queries come in as helpful. Let's set up a table like this:

```
>>> Group = Query()
>>> Permission = Query()
>>> groups = db.table('groups')
>>> groups.insert({
```

(continues on next page)

(continued from previous page)

```
        'name': 'user',
        'permissions': [{'type': 'read'}]})
>>> groups.insert({
        'name': 'sudo',
        'permissions': [{'type': 'read'}, {'type': 'sudo'}]})
>>> groups.insert({
        'name': 'admin',
        'permissions': [{'type': 'read'}, {'type': 'write'}, {'type': 'sudo'}]})
```

Now let's search this table using nested `any`/`all` queries:

```
>>> # Group has a permission with type 'read'
>>> groups.search(Group.permissions.any(Permission.type == 'read'))
[{'name': 'user', 'permissions': [{'type': 'read'}]},
 {'name': 'sudo', 'permissions': [{'type': 'read'}, {'type': 'sudo'}]},
 {'name': 'admin', 'permissions':
        [{'type': 'read'}, {'type': 'write'}, {'type': 'sudo'}]}]
>>> # Group has ONLY permission 'read'
>>> groups.search(Group.permissions.all(Permission.type == 'read'))
[{'name': 'user', 'permissions': [{'type': 'read'}]}]
```

As you can see, `any` tests if there is *at least one* document matching the query while `all` ensures *all* documents match the query.

The opposite operation, checking if a single item is contained in a list, is also possible using `one_of`:

```
>>> db.search(User.name.one_of(['jane', 'john']))
```

Query modifiers

TinyDB also allows you to use logical operations to modify and combine queries:

```
>>> # Negate a query:
>>> db.search(~ (User.name == 'John'))
```

```
>>> # Logical AND:
>>> db.search((User.name == 'John') & (User.age <= 30))
```

```
>>> # Logical OR:
>>> db.search((User.name == 'John') | (User.name == 'Bob'))
```

Note: When using `&` or `|`, make sure you wrap the conditions on both sides with parentheses or Python will mess up the comparison.

Also, when using negation (`~`) you'll have to wrap the query you want to negate in parentheses.

The reason for these requirements is that Python's binary operators that are used for query modifiers have a higher operator precedence than comparison operators. Simply put, `~ User.name == 'John'` is parsed by Python as `(~User.name) == 'John'` instead of `~(User.name == 'John')`. See also the Python docs on operator precedence for details.

Recap

Let's review the query operations we've learned:

Queries	
`Query().field.exists()`	Match any document where a field called `field` exists
`Query().field.matches(regex)`	Match any document with the whole field matching the regular expression
`Query().field.search(regex)`	Match any document with a substring of the field matching the regular expression
`Query().field.test(func, *args)`	Matches any document for which the function returns `True`
`Query().field.all(query | list)`	If given a query, matches all documents where all documents in the list `field` match the query. If given a list, matches all documents where all documents in the list `field` are a member of the given list
`Query().field.any(query | list)`	If given a query, matches all documents where at least one document in the list `field` match the query. If given a list, matches all documents where at least one documents in the list `field` are a member of the given list
`Query().field.one_of(list)`	Match if the field is contained in the list
Logical operations on queries	
`~ (query)`	Match documents that don't match the query
`(query1) & (query2)`	Match documents that match both queries
`(query1) | (query2)`	Match documents that match at least one of the queries

1.3.3 Handling Data

Next, let's look at some more ways to insert, update and retrieve data from your database.

Inserting data

As already described you can insert a document using `db.insert(...)`. In case you want to insert multiple documents, you can use `db.insert_multiple(...)`:

```
>>> db.insert_multiple([
        {'name': 'John', 'age': 22},
        {'name': 'John', 'age': 37}])
>>> db.insert_multiple({'int': 1, 'value': i} for i in range(2))
```

Also in some cases it may be useful to specify the document ID yourself when inserting data. You can do that by using the *Document* class:

```
>>> db.insert(Document({'name': 'John', 'age': 22}, doc_id=12))
12
```

Updating data

Sometimes you want to update all documents in your database. In this case, you can leave out the `query` argument:

```
>>> db.update({'foo': 'bar'})
```

When passing a dict to `db.update(fields, query)`, it only allows you to update a document by adding or overwriting its values. But sometimes you may need to e.g. remove one field or increment its value. In that case you can pass a function instead of `fields`:

```
>>> from tinydb.operations import delete
>>> db.update(delete('key1'), User.name == 'John')
```

This will remove the key `key1` from all matching documents. TinyDB comes with these operations:

- `delete(key)`: delete a key from the document
- `increment(key)`: increment the value of a key
- `decrement(key)`: decrement the value of a key
- `add(key, value)`: add `value` to the value of a key (also works for strings)
- `subtract(key, value)`: subtract `value` from the value of a key
- `set(key, value)`: set `key` to `value`

Of course you also can write your own operations:

```
>>> def your_operation(your_arguments):
...     def transform(doc):
...         # do something with the document
...         # ...
...     return transform
...
>>> db.update(your_operation(arguments), query)
```

In order to perform multiple update operations at once, you can use the `update_multiple` method like this:

```
>>> db.update_multiple([
...     ({'int': 2}, where('char') == 'a'),
...     ({'int': 4}, where('char') == 'b'),
... ])
```

You also can use mix normal updates with update operations:

```
>>> db.update_multiple([
...     ({'int': 2}, where('char') == 'a'),
...     ({delete('int'), where('char') == 'b'),
... ])
```

1.3.4 Data access and modification

Upserting data

In some cases you'll need a mix of both `update` and `insert`: `upsert`. This operation is provided a document and a query. If it finds any documents matching the query, they will be updated with the data from the provided document. On the other hand, if no matching document is found, it inserts the provided document into the table:

```
>>> db.upsert({'name': 'John', 'logged-in': True}, User.name == 'John')
```

This will update all users with the name John to have `logged-in` set to `True`. If no matching user is found, a new document is inserted with both the name set and the `logged-in` flag.

To use the ID of the document as matching criterion a *`Document`* with `doc_id` is passed instead of a query:

```
>>> db.upsert(Document({'name': 'John', 'logged-in': True}, doc_id=12))
```

Retrieving data

There are several ways to retrieve data from your database. For instance you can get the number of stored documents:

```
>>> len(db)
3
```

Hint: This will return the number of documents in the default table (see the notes on the *default table*).

Then of course you can use `db.search(...)` as described in the *Getting Started* section. But sometimes you want to get only one matching document. Instead of using

```
>>> try:
...     result = db.search(User.name == 'John')[0]
... except IndexError:
...     pass
```

you can use `db.get(...)`:

```
>>> db.get(User.name == 'John')
{'name': 'John', 'age': 22}
>>> db.get(User.name == 'Bobby')
None
```

Caution: If multiple documents match the query, probably a random one of them will be returned!

Often you don't want to search for documents but only know whether they are stored in the database. In this case `db.contains(...)` is your friend:

```
>>> db.contains(User.name == 'John')
```

In a similar manner you can look up the number of documents matching a query:

```
>>> db.count(User.name == 'John')
2
```

Recap

Let's summarize the ways to handle data:

Inserting data	
`db.insert_multiple(...)`	Insert multiple documents
Updating data	
`db.update(operation, ...)`	Update all matching documents with a special operation
Retrieving data	
`len(db)`	Get the number of documents in the database
`db.get(query)`	Get one document matching the query
`db.contains(query)`	Check if the database contains a matching document
`db.count(query)`	Get the number of matching documents

Note: This was a new feature in v3.6.0

1.3.5 Using Document IDs

Internally TinyDB associates an ID with every document you insert. It's returned after inserting a document:

```
>>> db.insert({'name': 'John', 'age': 22})
3
>>> db.insert_multiple([{...}, {...}, {...}])
[4, 5, 6]
```

In addition you can get the ID of already inserted documents using `document.doc_id`. This works both with `get` and `all`:

```
>>> el = db.get(User.name == 'John')
>>> el.doc_id
3
>>> el = db.all()[0]
>>> el.doc_id
1
>>> el = db.all()[-1]
>>> el.doc_id
12
```

Different TinyDB methods also work with IDs, namely: `update`, `remove`, `contains` and `get`. The first two also return a list of affected IDs.

```
>>> db.update({'value': 2}, doc_ids=[1, 2])
>>> db.contains(doc_id=1)
True
>>> db.remove(doc_ids=[1, 2])
>>> db.get(doc_id=3)
{...}
```

Using `doc_id` instead of `Query()` again is slightly faster in operation.

Recap

Let's sum up the way TinyDB supports working with IDs:

Getting a document's ID	
`db.insert(...)`	Returns the inserted document's ID
`db.insert_multiple(...)`	Returns the inserted documents' ID
`document.doc_id`	Get the ID of a document fetched from the db
Working with IDs	
`db.get(doc_id=...)`	Get the document with the given ID
`db.contains(doc_id=...)`	Check if the db contains a document with the given IDs
`db.update({...}, doc_ids=[...])`	Update all documents with the given IDs
`db.remove(doc_ids=[...])`	Remove all documents with the given IDs

1.3.6 Tables

TinyDB supports working with multiple tables. They behave just the same as the `TinyDB` class. To create and use a table, use `db.table(name)`.

```
>>> table = db.table('table_name')
>>> table.insert({'value': True})
>>> table.all()
[{'value': True}]
>>> for row in table:
>>>     print(row)
{'value': True}
```

To remove a table from a database, use:

```
>>> db.drop_table('table_name')
```

If on the other hand you want to remove all tables, use the counterpart:

```
>>> db.drop_tables()
```

Finally, you can get a list with the names of all tables in your database:

```
>>> db.tables()
{'_default', 'table_name'}
```

Default Table

TinyDB uses a table named `_default` as the default table. All operations on the database object (like `db.insert(...)`) operate on this table. The name of this table can be modified by setting the `default_table_name` class variable to modify the default table name for all instances:

```
>>> #1: for a single instance only
>>> db = TinyDB(storage=SomeStorage)
>>> db.default_table_name = 'my-default'
>>> #2: for all instances
>>> TinyDB.default_table_name = 'my-default'
```

Query Caching

TinyDB caches query result for performance. That way re-running a query won't have to read the data from the storage as long as the database hasn't been modified. You can optimize the query cache size by passing the `cache_size` to the `table(...)` function:

```
>>> table = db.table('table_name', cache_size=30)
```

Hint: You can set `cache_size` to `None` to make the cache unlimited in size. Also, you can set `cache_size` to 0 to disable it.

Hint: The TinyDB query cache doesn't check if the underlying storage that the database uses has been modified by an external process. In this case the query cache may return outdated results. To clear the cache and read data from the storage again you can use `db.clear_cache()`.

Hint: When using an unlimited cache size and `test()` queries, TinyDB will store a reference to the test function. As a result of that behavior long-running applications that use `lambda` functions as a test function may experience memory leaks.

1.3.7 Storage & Middleware

Storage Types

TinyDB comes with two storage types: JSON and in-memory. By default TinyDB stores its data in JSON files so you have to specify the path where to store it:

```
>>> from tinydb import TinyDB, where
>>> db = TinyDB('path/to/db.json')
```

To use the in-memory storage, use:

```
>>> from tinydb.storages import MemoryStorage
>>> db = TinyDB(storage=MemoryStorage)
```

Hint: All arguments except for the `storage` argument are forwarded to the underlying storage. For the JSON storage you can use this to pass additional keyword arguments to Python's json.dump(...) method. For example, you can set it to create prettified JSON files like this:

```
>>> db = TinyDB('db.json', sort_keys=True, indent=4, separators=(',', ': '))
```

To modify the default storage for all `TinyDB` instances, set the `default_storage_class` class variable:

```
>>> TinyDB.default_storage_class = MemoryStorage
```

In case you need to access the storage instance directly, you can use the `storage` property of your TinyDB instance. This may be useful to call method directly on the storage or middleware:

```
>>> db = TinyDB(storage=CachingMiddleware(MemoryStorage))
<tinydb.middlewares.CachingMiddleware at 0x10991def0>
>>> db.storage.flush()
```

Middleware

Middleware wraps around existing storage allowing you to customize their behaviour.

```
>>> from tinydb.storages import JSONStorage
>>> from tinydb.middlewares import CachingMiddleware
>>> db = TinyDB('/path/to/db.json', storage=CachingMiddleware(JSONStorage))
```

Hint: You can nest middleware:

```
>>> db = TinyDB('/path/to/db.json',
                storage=FirstMiddleware(SecondMiddleware(JSONStorage)))
```

CachingMiddleware

The `CachingMiddleware` improves speed by reducing disk I/O. It caches all read operations and writes data to disk after a configured number of write operations.

To make sure that all data is safely written when closing the table, use one of these ways:

```
# Using a context manager:
with database as db:
    # Your operations
```

```
# Using the close function
db.close()
```

1.3.8 MyPy Type Checking

TinyDB comes with type annotations that MyPy can use to make sure you're using the API correctly. Unfortunately, MyPy doesn't understand all code patterns that TinyDB uses. For that reason TinyDB ships a MyPy plugin that helps correctly type checking code that uses TinyDB. To use it, add it to the plugins list in the MyPy configuration file (typically located in `setup.cfg` or `mypy.ini`):

```
[mypy]
plugins = tinydb.mypy_plugin
```

1.3.9 What's next

Congratulations, you've made through the user guide! Now go and build something awesome or dive deeper into TinyDB with these resources:

- Want to learn how to customize TinyDB (storages, middlewares) and what extensions exist? Check out *How to Extend TinyDB* and *Extensions*.

- Want to study the API in detail? Read *API Documentation*.
- Interested in contributing to the TinyDB development guide? Go on to the *Contribution Guidelines*.

CHAPTER 2

Extending TinyDB

2.1 How to Extend TinyDB

There are three main ways to extend TinyDB and modify its behaviour:

1. custom storages,
2. custom middlewares,
3. use hooks and overrides, and
4. subclassing `TinyDB` and `Table`.

Let's look at them in this order.

2.1.1 Write a Custom Storage

First, we have support for custom storages. By default TinyDB comes with an in-memory storage and a JSON file storage. But of course you can add your own. Let's look how you could add a YAML storage using PyYAML:

```
import yaml

class YAMLStorage(Storage):
    def __init__(self, filename):  # (1)
        self.filename = filename

    def read(self):
        with open(self.filename) as handle:
            try:
                data = yaml.safe_load(handle.read())  # (2)
                return data
            except yaml.YAMLError:
                return None  # (3)
```

(continues on next page)

(continued from previous page)

```
    def write(self, data):
        with open(self.filename, 'w+') as handle:
            yaml.dump(data, handle)

    def close(self):  # (4)
        pass
```

There are some things we should look closer at:

1. The constructor will receive all arguments passed to TinyDB when creating the database instance (except `storage` which TinyDB itself consumes). In other words calling `TinyDB('something', storage=YAMLStorage)` will pass `'something'` as an argument to `YAMLStorage`.
2. We use `yaml.safe_load` as recommended by the PyYAML documentation when processing data from a potentially untrusted source.
3. If the storage is uninitialized, TinyDB expects the storage to return `None` so it can do any internal initialization that is necessary.
4. If your storage needs any cleanup (like closing file handles) before an instance is destroyed, you can put it in the `close()` method. To run these, you'll either have to run `db.close()` on your `TinyDB` instance or use it as a context manager, like this:

   ```
   with TinyDB('db.yml', storage=YAMLStorage) as db:
       # ...
   ```

Finally, using the YAML storage is very straight-forward:

```
db = TinyDB('db.yml', storage=YAMLStorage)
# ...
```

2.1.2 Write Custom Middleware

Sometimes you don't want to write a new storage module but rather modify the behaviour of an existing one. As an example we'll build middleware that filters out empty items.

Because middleware acts as a wrapper around a storage, they needs a `read()` and a `write(data)` method. In addition, they can access the underlying storage via `self.storage`. Before we start implementing we should look at the structure of the data that the middleware receives. Here's what the data that goes through the middleware looks like:

```
{
    '_default': {
        1: {'key': 'value'},
        2: {'key': 'value'},
        # other items
    },
    # other tables
}
```

Thus, we'll need two nested loops:

1. Process every table
2. Process every item

Now let's implement that:

```
class RemoveEmptyItemsMiddleware(Middleware):
    def __init__(self, storage_cls):
        # Any middleware *has* to call the super constructor
        # with storage_cls
        super(self).__init__(storage_cls)  # (1)

    def read(self):
        data = self.storage.read()

        for table_name in data:
            table_data = data[table_name]

            for doc_id in table:
                item = table_data[doc_id]

                if item == {}:
                    del table_data[doc_id]

        return data

    def write(self, data):
        for table_name in data:
            table_data = data[table_name]

            for doc_id in table:
                item = table_data[doc_id]

                if item == {}:
                    del table_data[doc_id]

        self.storage.write(data)

    def close(self):
        self.storage.close()
```

Note that the constructor calls the middleware constructor (1) and passes the storage class to the middleware constructor.

To wrap storage with this new middleware, we use it like this:

```
db = TinyDB(storage=RemoveEmptyItemsMiddleware(SomeStorageClass))
```

Here `SomeStorageClass` should be replaced with the storage you want to use. If you leave it empty, the default storage will be used (which is the `JSONStorage`).

2.1.3 Use hooks and overrides

There are cases when neither creating a custom storage nor using a custom middlware will allow you to adapt TinyDB in the way you need. In this case you can modify TinyDB's behavior by using predefined hooks and override points. For example you can configure the name of the default table by setting `TinyDB.default_table_name`:

```
TinyDB.default_table_name = 'my_table_name'
```

Both *TinyDB* and the *Table* classes allow modifying their behavior using hooks and overrides. To use `Table`'s overrides, you can access the class using `TinyDB.table_class`:

```
TinyDB.table_class.default_query_cache_capacity = 100
```

Read the *API Documentation* for more details on the available hooks and override points.

2.1.4 Subclassing `TinyDB` and `Table`

Finally, there's the last option to modify TinyDB's behavior. That way you can change how TinyDB itself works more deeply than using the other extension mechanisms.

When creating a subclass you can use it by using hooks and overrides to override the default classes that TinyDB uses:

```
class MyTable(Table):
    # Add your method overrides
    ...

TinyDB.table_class = MyTable

# Continue using TinyDB as usual
```

TinyDB's source code is documented with extensions in mind, explaining how everything works even for internal methods and classes. Feel free to dig into the source and adapt everything you need for your projects.

2.2 Extensions

Here are some extensions that might be useful to you:

2.2.1 `tinyindex`

Repo: https://github.com/eugene-eeo/tinyindex
Status: *experimental*
Description: Document indexing for TinyDB. Basically ensures deterministic (as long as there aren't any changes to the table) yielding of documents.

2.2.2 `tinymongo`

Repo: https://github.com/schapman1974/tinymongo
Status: *experimental*
Description: A simple wrapper that allows to use TinyDB as a flat file drop-in replacement for MongoDB.

2.2.3 `TinyMP`

Repo: https://github.com/alshapton/TinyMP
Status: *stable*
Description: A MessagePack-based storage extension to tinydb using http://msgpack.org

2.2.4 tinyrecord

Repo: https://github.com/eugene-eeo/tinyrecord
Status: *stable*
Description: Tinyrecord is a library which implements experimental atomic transaction support for the TinyDB NoSQL database. It uses a record-first then execute architecture which allows us to minimize the time that we are within a thread lock.

2.2.5 tinydb-appengine

Repo: https://github.com/imalento/tinydb-appengine
Status: *stable*
Description: `tinydb-appengine` provides TinyDB storage for App Engine. You can use JSON readonly.

2.2.6 tinydb-serialization

Repo: https://github.com/msiemens/tinydb-serialization
Status: *stable*
Description: `tinydb-serialization` provides serialization for objects that TinyDB otherwise couldn't handle.

2.2.7 tinydb-smartcache

Repo: https://github.com/msiemens/tinydb-smartcache
Status: *stable*
Description: `tinydb-smartcache` provides a smart query cache for TinyDB. It updates the query cache when inserting/removing/updating documents so the cache doesn't get invalidated. It's useful if you perform lots of queries while the data changes only little.

2.2.8 aiotinydb

Repo: https://github.com/ASMfreaK/aiotinydb
Status: *stable*
Description: asyncio compatibility shim for TinyDB. Enables usage of TinyDB in asyncio-aware contexts without slow syncronous IO.

2.2.9 TinyDBTimestamps

Repo: https://github.com/pachacamac/TinyDBTimestamps
Status: *experimental*
Description: Automatically add create at/ update at timestamps to TinyDB documents.

CHAPTER 3

API Reference

3.1 API Documentation

3.1.1 tinydb.database

class tinydb.database.**TinyDB**(**args*, ***kwargs*)

The main class of TinyDB.

The `TinyDB` class is responsible for creating the storage class instance that will store this database's documents, managing the database tables as well as providing access to the default table.

For table management, a simple `dict` is used that stores the table class instances accessible using their table name.

Default table access is provided by forwarding all unknown method calls and property access operations to the default table by implementing `__getattr__`.

When creating a new instance, all arguments and keyword arguments (except for `storage`) will be passed to the storage class that is provided. If no storage class is specified, *JSONStorage* will be used.

Customization

For customization, the following class variables can be set:

- `table_class` defines the class that is used to create tables,
- `default_table_name` defines the name of the default table, and
- `default_storage_class` will define the class that will be used to create storage instances if no other storage is passed.

New in version 4.0.

Data Storage Model

Data is stored using a storage class that provides persistence for a `dict` instance. This `dict` contains all tables and their data. The data is modelled like this:

```
{
    'table1': {
        0: {document...},
        1: {document...},
    },
    'table2': {
        ...
    }
}
```

Each entry in this `dict` uses the table name as its key and a `dict` of documents as its value. The document `dict` contains document IDs as keys and the documents themselves as values.

Parameters **storage** – The class of the storage to use. Will be initialized with `args` and `kwargs`.

table_class
alias of *tinydb.table.Table*

default_table_name = '_default'
The name of the default table

New in version 4.0.

default_storage_class
alias of *tinydb.storages.JSONStorage*

table(*name: str*, ***kwargs*) → tinydb.table.Table
Get access to a specific table.

If the table hasn't been accessed yet, a new table instance will be created using the *table_class* class. Otherwise, the previously created table instance wil be returned.

All futher options besides the name are passed to the table class which by default is *Table*. Check its documentation for further parameters you can pass.

Parameters

- **name** – The name of the table.
- **kwargs** – Keyword arguments to pass to the table class constructor

tables() → Set[str]
Get the names of all tables in the database.

Returns a set of table names

drop_tables() → None
Drop all tables from the database. **CANNOT BE REVERSED!**

drop_table(*name: str*) → None
Drop a specific table from the database. **CANNOT BE REVERSED!**

Parameters **name** – The name of the table to drop.

storage
Get the storage instance used for this TinyDB instance.

Returns This instance's storage

Return type *Storage*

close() → None
Close the database.

This may be needed if the storage instance used for this database needs to perform cleanup operations like closing file handles.

To ensure this method is called, the TinyDB instance can be used as a context manager:

```
with TinyDB('data.json') as db:
    db.insert({'foo': 'bar'})
```

Upon leaving this context, the `close` method will be called.

3.1.2 tinydb.table

class tinydb.table.**Table**(*storage: tinydb.storages.Storage*, *name: str*, *cache_size: int = 10*)
Represents a single TinyDB table.

It provides methods for accessing and manipulating documents.

Query Cache

As an optimization, a query cache is implemented using a *LRUCache*. This class mimics the interface of a normal `dict`, but starts to remove the least-recently used entries once a threshold is reached.

The query cache is updated on every search operation. When writing data, the whole cache is discareded as the query results may have changed.

Customization

For customization, the following class variables can be set:

- `document_class` defines the class that is used to represent documents,
- `document_id_class` defines the class that is used to represent document IDs,
- `query_cache_class` defines the class that is used for the query cache
- `default_query_cache_capacity` defines the default capacity of the query cache

New in version 4.0.

Parameters

- **storage** – The storage instance to use for this table
- **name** – The table name
- **cache_size** – Maximum capacity of query cache

document_class
The class used to represent documents

New in version 4.0.

alias of *Document*

document_id_class
alias of builtins.int

query_cache_class
alias of *tinydb.utils.LRUCache*

default_query_cache_capacity = 10
The default capacity of the query cache

New in version 4.0.

__init__(*storage: tinydb.storages.Storage*, *name: str*, *cache_size: int = 10*)
Create a table instance.

__repr__()
Return repr(self).

name
Get the table name.

storage
Get the table storage instance.

insert(*document: Mapping[KT, VT_co]*) → int
Insert a new document into the table.

Parameters **document** – the document to insert

Returns the inserted document's ID

insert_multiple(*documents: Iterable[Mapping[KT, VT_co]]*) → List[int]
Insert multiple documents into the table.

Parameters **documents** – a Iterable of documents to insert

Returns a list containing the inserted documents' IDs

all() → List[tinydb.table.Document]
Get all documents stored in the table.

Returns a list with all documents.

search(*cond: tinydb.queries.QueryLike*) → List[tinydb.table.Document]
Search for all documents matching a 'where' cond.

Parameters **cond** – the condition to check against

Returns list of matching documents

get(*cond: Optional[tinydb.queries.QueryLike] = None*, *doc_id: Optional[int] = None*) → Optional[tinydb.table.Document]
Get exactly one document specified by a query or a document ID.

Returns None if the document doesn't exist.

Parameters

- **cond** – the condition to check against
- **doc_id** – the document's ID

Returns the document or None

contains(*cond: Optional[tinydb.queries.QueryLike] = None*, *doc_id: Optional[int] = None*) → bool
Check whether the database contains a document matching a query or an ID.

If doc_id is set, it checks if the db contains the specified ID.

Parameters

- **cond** – the condition use
- **doc_id** – the document ID to look for

update(*fields: Union[Mapping[KT, VT_co], Callable[[Mapping[KT, VT_co]], None]], cond: Optional[tinydb.queries.QueryLike] = None, doc_ids: Optional[Iterable[int]] = None*) → List[int]

Update all matching documents to have a given set of fields.

Parameters

- **fields** – the fields that the matching documents will have or a method that will update the documents
- **cond** – which documents to update
- **doc_ids** – a list of document IDs

Returns a list containing the updated document's ID

update_multiple(*updates: Iterable[Tuple[Union[Mapping[KT, VT_co], Callable[[Mapping[KT, VT_co]], None]], tinydb.queries.QueryLike]]*) → List[int]

Update all matching documents to have a given set of fields.

Returns a list containing the updated document's ID

upsert(*document: Mapping[KT, VT_co], cond: Optional[tinydb.queries.QueryLike] = None*) → List[int]

Update documents, if they exist, insert them otherwise.

Note: This will update *all* documents matching the query. Document argument can be a tinydb.table.Document object if you want to specify a doc_id.

Parameters

- **document** – the document to insert or the fields to update
- **cond** – which document to look for, optional if you've passed a

Document with a doc_id :returns: a list containing the updated documents' IDs

remove(*cond: Optional[tinydb.queries.QueryLike] = None*, *doc_ids: Optional[Iterable[int]] = None*) → List[int]

Remove all matching documents.

Parameters

- **cond** – the condition to check against
- **doc_ids** – a list of document IDs

Returns a list containing the removed documents' ID

truncate() → None

Truncate the table by removing all documents.

count(*cond: tinydb.queries.QueryLike*) → int

Count the documents matching a query.

Parameters **cond** – the condition use

clear_cache() → None

Clear the query cache.

__len__()

Count the total number of documents in this table.

__iter__() → Iterator[tinydb.table.Document]
Iterate over all documents stored in the table.

Returns an iterator over all documents.

class tinydb.table.**Document**(*value: Mapping[KT, VT_co], doc_id: int*)
A document stored in the database.

This class provides a way to access both a document's content as well as its ID using `doc.doc_id`.

doc_id
The document's id

__init__(*value: Mapping[KT, VT_co], doc_id: int*)
Initialize self. See help(type(self)) for accurate signature.

3.1.3 tinydb.queries

class tinydb.queries.**Query**
TinyDB Queries.

Allows to build queries for TinyDB databases. There are two main ways of using queries:

1) ORM-like usage:

```
>>> User = Query()
>>> db.search(User.name == 'John Doe')
>>> db.search(User['logged-in'] == True)
```

2) Classical usage:

```
>>> db.search(where('value') == True)
```

Note that `where(...)` is a shorthand for `Query(...)` allowing for a more fluent syntax.

Besides the methods documented here you can combine queries using the binary AND and OR operators:

```
>>> # Binary AND:
>>> db.search((where('field1').exists()) & (where('field2') == 5))
>>> # Binary OR:
>>> db.search((where('field1').exists()) | (where('field2') == 5))
```

Queries are executed by calling the resulting object. They expect to get the document to test as the first argument and return `True` or `False` depending on whether the documents matches the query or not.

__init__() → None
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__hash__()
Return hash(self).

__eq__(*rhs: Any*)
Test a dict value for equality.

```
>>> Query().f1 == 42
```

Parameters **rhs** – The value to compare against

__ne__(*rhs: Any*)
Test a dict value for inequality.

```
>>> Query().f1 != 42
```

Parameters **rhs** – The value to compare against

__lt__(*rhs: Any*) → tinydb.queries.QueryInstance
Test a dict value for being lower than another value.

```
>>> Query().f1 < 42
```

Parameters **rhs** – The value to compare against

__le__(*rhs: Any*) → tinydb.queries.QueryInstance
Test a dict value for being lower than or equal to another value.

```
>>> where('f1') <= 42
```

Parameters **rhs** – The value to compare against

__gt__(*rhs: Any*) → tinydb.queries.QueryInstance
Test a dict value for being greater than another value.

```
>>> Query().f1 > 42
```

Parameters **rhs** – The value to compare against

__ge__(*rhs: Any*) → tinydb.queries.QueryInstance
Test a dict value for being greater than or equal to another value.

```
>>> Query().f1 >= 42
```

Parameters **rhs** – The value to compare against

exists() → tinydb.queries.QueryInstance
Test for a dict where a provided key exists.

```
>>> Query().f1.exists()
```

matches(*regex: str*, *flags: int = 0*) → tinydb.queries.QueryInstance
Run a regex test against a dict value (whole string has to match).

```
>>> Query().f1.matches(r'^\w+$')
```

Parameters

- **regex** – The regular expression to use for matching
- **flags** – regex flags to pass to `re.match`

search(*regex: str*, *flags: int = 0*) → tinydb.queries.QueryInstance
Run a regex test against a dict value (only substring string has to match).

```
>>> Query().f1.search(r'^\w+$')
```

Parameters

- **regex** – The regular expression to use for matching
- **flags** – regex flags to pass to `re.match`

test(*func: Callable[[Mapping[KT, VT_co]], bool]*, **args*) → tinydb.queries.QueryInstance
Run a user-defined test function against a dict value.

```
>>> def test_func(val):
...     return val == 42
...
>>> Query().f1.test(test_func)
```

Warning: The test fuction provided needs to be deterministic (returning the same value when provided with the same arguments), otherwise this may mess up the query cache that *Table* implements.

Parameters

- **func** – The function to call, passing the dict as the first argument
- **args** – Additional arguments to pass to the test function

any(*cond: Union[tinydb.queries.QueryInstance, List[Any]]*) → tinydb.queries.QueryInstance
Check if a condition is met by any document in a list, where a condition can also be a sequence (e.g. list).

```
>>> Query().f1.any(Query().f2 == 1)
```

Matches:

```
{'f1': [{'f2': 1}, {'f2': 0}]}
```

```
>>> Query().f1.any([1, 2, 3])
```

Matches:

```
{'f1': [1, 2]}
{'f1': [3, 4, 5]}
```

Parameters **cond** – Either a query that at least one document has to match or a list of which at least one document has to be contained in the tested document.

all(*cond: Union[QueryInstance, List[Any]]*) → tinydb.queries.QueryInstance
Check if a condition is met by all documents in a list, where a condition can also be a sequence (e.g. list).

```
>>> Query().f1.all(Query().f2 == 1)
```

Matches:

```
{'f1': [{'f2': 1}, {'f2': 1}]}
```

```
>>> Query().f1.all([1, 2, 3])
```

Matches:

```
{'f1': [1, 2, 3, 4, 5]}
```

Parameters **cond** – Either a query that all documents have to match or a list which has to be contained in the tested document.

one_of(*items: List[Any]*) → tinydb.queries.QueryInstance
Check if the value is contained in a list or generator.

```
>>> Query().f1.one_of(['value 1', 'value 2'])
```

Parameters **items** – The list of items to check with

noop() → tinydb.queries.QueryInstance
Always evaluate to `True`.

Useful for having a base value when composing queries dynamically.

class tinydb.queries.**QueryInstance**(*test: Callable[[Mapping[KT, VT_co]], bool], hashval: Tuple*)
A query instance.

This is the object on which the actual query operations are performed. The *Query* class acts like a query builder and generates *QueryInstance* objects which will evaluate their query against a given document when called.

Query instances can be combined using logical OR and AND and inverted using logical NOT.

In order to be usable in a query cache, a query needs to have a stable hash value with the same query always returning the same hash. That way a query instance can be used as a key in a dictionary.

__init__(*test: Callable[[Mapping[KT, VT_co]], bool], hashval: Tuple*)
Initialize self. See help(type(self)) for accurate signature.

__call__(*value: Mapping[KT, VT_co]*) → bool
Evaluate the query to check if it matches a specified value.

Parameters **value** – The value to check.

Returns Wether the value matchs this query.

__hash__()
Return hash(self).

__repr__()
Return repr(self).

__eq__(*other: object*)
Return self==value.

3.1.4 tinydb.operations

A collection of update operations for TinyDB.

They are used for updates like this:

```
>>> db.update(delete('foo'), where('foo') == 2)
```

This would delete the `foo` field from all documents where `foo` equals 2.

`tinydb.operations.`**`delete`**(*field*)
Delete a given field from the document.

`tinydb.operations.`**`add`**(*field*, *n*)
Add `n` to a given field in the document.

`tinydb.operations.`**`subtract`**(*field*, *n*)
Substract `n` to a given field in the document.

`tinydb.operations.`**`set`**(*field*, *val*)
Set a given field to `val`.

`tinydb.operations.`**`increment`**(*field*)
Increment a given field in the document by 1.

`tinydb.operations.`**`decrement`**(*field*)
Decrement a given field in the document by 1.

3.1.5 `tinydb.storage`

Contains the *base class* for storages and implementations.

class `tinydb.storages.`**`Storage`**
The abstract base class for all Storages.

A Storage (de)serializes the current state of the database and stores it in some place (memory, file on disk, ...).

`read`()
Read the last stored state.

`write`(*data*)
Write the current state of the database to the storage.

`close`()
Optional: Close open file handles, etc.

class `tinydb.storages.`**`JSONStorage`**(*path: str*, *create_dirs=False*, *encoding=None*, *access_mode='r+'*, ***kwargs*)
Store the data in a JSON file.

`__init__`(*path: str*, *create_dirs=False*, *encoding=None*, *access_mode='r+'*, ***kwargs*)
Create a new instance.

Also creates the storage file, if it doesn't exist and the access mode is appropriate for writing.

Parameters

- **`path`** – Where to store the JSON data.
- **`access_mode`** (*str*) – mode in which the file is opened (r, r+, w, a, x, b, t, +, U)

`close`() → None
Optional: Close open file handles, etc.

`read`() → Optional[Dict[str, Dict[str, Any]]]
Read the current state.

Any kind of deserialization should go here.

Return `None` here to indicate that the storage is empty.

write(*data: Dict[str, Dict[str, Any]]*)
Write the current state of the database to the storage.

Any kind of serialization should go here.

Parameters **data** – The current state of the database.

class tinydb.storages.**MemoryStorage**
Store the data as JSON in memory.

__init__()
Create a new instance.

read() → Optional[Dict[str, Dict[str, Any]]]
Read the current state.

Any kind of deserialization should go here.

Return `None` here to indicate that the storage is empty.

write(*data: Dict[str, Dict[str, Any]]*)
Write the current state of the database to the storage.

Any kind of serialization should go here.

Parameters **data** – The current state of the database.

3.1.6 tinydb.middlewares

Contains the *base class* for middlewares and implementations.

class tinydb.middlewares.**Middleware**
The base class for all Middlewares.

Middlewares hook into the read/write process of TinyDB allowing you to extend the behaviour by adding caching, logging, ...

If `read()` or `write()` are not overloaded, they will be forwarded directly to the storage instance.

storage

Type *Storage*

Access to the underlying storage instance.

read()
Read the last stored state.

write(*data*)
Write the current state of the database to the storage.

close()
Optional: Close open file handles, etc.

class tinydb.middlewares.**CachingMiddleware**(*storage_cls*)
Add some caching to TinyDB.

This Middleware aims to improve the performance of TinyDB by writing only the last DB state every *WRITE_CACHE_SIZE* time and reading always from cache.

WRITE_CACHE_SIZE = 1000
The number of write operations to cache before writing to disc

__init__(*storage_cls*)
Initialize self. See help(type(self)) for accurate signature.

flush()
Flush all unwritten data to disk.

3.1.7 tinydb.utils

class tinydb.utils.**LRUCache**(*capacity=None*)
A least-recently used (LRU) cache with a fixed cache size.

This class acts as a dictionary but has a limited size. If the number of entries in the cache exeeds the cache size, the leat-recently accessed entry will be discareded.

This is implemented using an `OrderedDict`. On every access the accessed entry is moved to the front by re-inserting it into the `OrderedDict`. When adding an entry and the cache size is exceeded, the last entry will be discareded.

__init__(*capacity=None*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

CHAPTER 4

Additional Notes

4.1 Contribution Guidelines

Whether reporting bugs, discussing improvements and new ideas or writing extensions: Contributions to TinyDB are welcome! Here's how to get started:

1. Check for open issues or open a fresh issue to start a discussion around a feature idea or a bug
2. Fork the repository on Github, create a new branch off the *master* branch and start making your changes (known as GitHub Flow)
3. Write a test which shows that the bug was fixed or that the feature works as expected
4. Send a pull request and bug the maintainer until it gets merged and published :)

4.1.1 Philosophy of TinyDB

TinyDB aims to be simple and fun to use. Therefore two key values are simplicity and elegance of interfaces and code. These values will contradict each other from time to time. In these cases , try using as little magic as possible. In any case don't forget documenting code that isn't clear at first glance.

4.1.2 Code Conventions

In general the TinyDB source should always follow PEP 8. Exceptions are allowed in well justified and documented cases. However we make a small exception concerning docstrings:

When using multiline docstrings, keep the opening and closing triple quotes on their own lines and add an empty line after it.

```
def some_function():
    """
    Documentation ...
    """
```

(continues on next page)

(continued from previous page)

```
    # implementation ...
```

4.1.3 Version Numbers

TinyDB follows the SemVer versioning guidelines. This implies that backwards incompatible changes in the API will increment the major version. So think twice before making such changes.

4.2 Changelog

4.2.1 Version Numbering

TinyDB follows the SemVer versioning guidelines. For more information, see semver.org

4.2.2 unreleased

- *nothing yet*

4.2.3 v4.5.2 (2021-09-23)

- Fix: Make `Table.delete()`'s argument priorities consistent with other table methods. This means that if you pass both `cond` as well as `doc_ids` to `Table.delete()`, the latter will be prefered (see issue 424)

4.2.4 v4.5.1 (2021-07-17)

- Fix: Correctly install `typing-extensions` on Python 3.7 (see issue 413)

4.2.5 v4.5.0 (2021-06-25)

- Feature: Better type hinting/IntelliSense for PyCharm, VS Code and MyPy (see issue 372). PyCharm and VS Code should work out of the box, for MyPy see *MyPy Type Checking*

4.2.6 v4.4.0 (2021-02-11)

- Feature: Add operation for searching for all documents that match a `dict` fragment (see issue 300)
- Fix: Correctly handle queries that use fields that are also Query methods, e.g. `Query()['test']` for searching for documents with a `test` field (see issue 373)

4.2.7 v4.3.0 (2020-11-14)

- Feature: Add operation for updating multiple documents: `update_multiple` (see issue 346)
- Improvement: Expose type information for MyPy typechecking (PEP 561) (see pull request 352)

4.2.8 v4.2.0 (2020-10-03)

- Feature: Add support for specifying document IDs during insertion (see issue 303)
- Internal change: Use `OrderedDict.move_to_end()` in the query cache (see issue 338)

4.2.9 v4.1.1 (2020-05-08)

- Fix: Don't install dev-dependencies when installing from PyPI (see issue 315)

4.2.10 v4.1.0 (2020-05-07)

- Feature: Add a no-op query `Query().noop()` (see issue 313)
- Feature: Add a `access_mode` flag to `JSONStorage` to allow opening files read-only (see issue 297)
- Fix: Don't drop the first document that's being inserted when inserting data on an existing database (see issue 314)

4.2.11 v4.0.0 (2020-05-02)

Upgrade Notes

Breaking Changes

- Python 2 support has been removed, see issue 284 for background
- API changes:
  - Removed classes: `DataProxy`, `StorageProxy`
  - Attributes removed from `TinyDB` in favor of customizing `TinyDB`'s behavior by subclassing it and overloading `__init__(...)` and `table(...)`:
    * `DEFAULT_TABLE`
    * `DEFAULT_TABLE_KWARGS`
    * `DEFAULT_STORAGE`
  - Arguments removed from `TinyDB(...)`:
    * `default_table`: replace with `TinyDB.default_table_name = 'name'`
    * `table_class`: replace with `TinyDB.table_class = Class`
  - `TinyDB.contains(...)`'s `doc_ids` parameter has been renamed to `doc_id` and now only takes a single document ID
  - `TinyDB.purge_tables(...)` has been renamed to `TinyDB.drop_tables(...)`
  - `TinyDB.purge_table(...)` has been renamed to `TinyDB.drop_table(...)`
  - `TinyDB.write_back(...)` has been removed
  - `TinyDB.process_elements(...)` has been removed
  - `Table.purge()` has been renamed to `Table.truncate()`

  - Evaluating an empty `Query()` without any test operators will now result in an exception, use `Query().noop()` (introduced in v4.1.0) instead

- `ujson` support has been removed, see issue 263 and issue 306 for background
- The deprecated Element ID API has been removed (e.g. using the `Element` class or `eids` parameter) in favor the Document API, see pull request 158 for details on the replacement

Improvements

- TinyDB's internal architecture has been reworked to be more simple and streamlined in order to make it easier to customize TinyDB's behavior
- With the new architecture, TinyDB performance will improve for many applications

Bugfixes

- Don't break the tests when `ujson` is installed (see issue 262)
- Fix performance when reading data (see issue 250)
- Fix inconsistent purge function names (see issue 103)

4.2.12 v3.15.1 (2019-10-26)

- Internal change: fix missing values handling for `LRUCache`

4.2.13 v3.15.0 (2019-10-12)

- Feature: allow setting the parameters of TinyDB's default table (see issue 278)

4.2.14 v3.14.2 (2019-09-13)

- Internal change: support correct iteration for `LRUCache` objects

4.2.15 v3.14.1 (2019-07-03)

- Internal change: fix Query class to permit subclass creation (see pull request 270)

4.2.16 v3.14.0 (2019-06-18)

- Change: support for `ujson` is now deprecated (see issue 263)

4.2.17 v3.13.0 (2019-03-16)

- Feature: direct access to a TinyDB instance's storage (see issue 258)

4.2.18 v3.12.2 (2018-12-12)

- Internal change: convert documents to dicts during insertion (see pull request 256)
- Internal change: use tuple literals instead of tuple class/constructor (see pull request 247)
- Infra: ensure YAML tests are run (see pull request 252)

4.2.19 v3.12.1 (2018-11-09)

- Fix: Don't break when searching the same query multiple times (see pull request 249)
- Internal change: allow `collections.abc.Mutable` as valid document types (see pull request 245)

4.2.20 v3.12.0 (2018-11-06)

- Feature: Add encoding option to `JSONStorage` (see pull request 238)
- Internal change: allow `collections.abc.Mutable` as valid document types (see pull request 245)

4.2.21 v3.11.1 (2018-09-13)

- Bugfix: Make path queries (`db.search(where('key'))`) work again (see issue 232)
- Improvement: Add custom `repr` representations for main classes (see pull request 229)

4.2.22 v3.11.0 (2018-08-20)

- **Drop official support for Python 3.3**. Python 3.3 has reached its official End Of Life as of September 29, 2017. It will probably continue to work, but will not be tested against (issue 217)
- Feature: Allow extending TinyDB with a custom storage proxy class (see pull request 224)
- Bugfix: Return list of document IDs for upsert when creating a new document (see issue 223)

4.2.23 v3.10.0 (2018-07-21)

- Feature: Add support for regex flags (see pull request 216)

4.2.24 v3.9.0 (2018-04-24)

- Feature: Allow setting a table class for single table only (see issue 197)
- Internal change: call fsync after flushing `JSONStorage` (see issue 208)

4.2.25 v3.8.1 (2018-03-26)

- Bugfix: Don't install tests as a package anymore (see pull request #195)

4.2.26 v3.8.0 (2018-03-01)

- Feature: Allow disabling the query cache with `db.table(name, cache_size=0)` (see pull request #187)
- Feature: Add `db.write_back(docs)` for replacing documents (see pull request #184)

4.2.27 v3.7.0 (2017-11-11)

- Feature: `one_of` for checking if a value is contained in a list (see issue 164)
- Feature: Upsert (insert if document doesn't exist, otherwise update; see https://forum.m-siemens.de/d/30-primary-key-well-sort-of)
- Internal change: don't read from storage twice during initialization (see https://forum.m-siemens.de/d/28-reads-the-whole-data-file-twice)

4.2.28 v3.6.0 (2017-10-05)

- Allow updating all documents using `db.update(fields)` (see issue #157).
- Rename elements to documents. Document IDs now available with `doc.doc_id`, using `doc.eid` is now deprecated (see pull request #158)

4.2.29 v3.5.0 (2017-08-30)

- Expose the table name via `table.name` (see issue #147).
- Allow better subclassing of the `TinyDB` class (see pull request #150).

4.2.30 v3.4.1 (2017-08-23)

- Expose TinyDB version via `import tinyb; tinydb.__version__` (see issue #148).

4.2.31 v3.4.0 (2017-08-08)

- Add new update operations: `add(key, value)`, `substract(key, value)`, and `set(key, value)` (see pull request #145).

4.2.32 v3.3.1 (2017-06-27)

- Use relative imports to allow vendoring TinyDB in other packages (see pull request #142).

4.2.33 v3.3.0 (2017-06-05)

- Allow iterating over a database or table yielding all documents (see pull request #139).

4.2.34 v3.2.3 (2017-04-22)

- Fix bug with accidental modifications to the query cache when modifying the list of search results (see issue #132).

4.2.35 v3.2.2 (2017-01-16)

- Fix the `Query` constructor to prevent wrong usage (see issue #117).

4.2.36 v3.2.1 (2016-06-29)

- Fix a bug with queries on documents that have a `path` key (see pull request #107).
- Don't write to the database file needlessly when opening the database (see pull request #104).

4.2.37 v3.2.0 (2016-04-25)

- Add a way to specify the default table name via *default_table* (see pull request #98).
- Add `db.purge_table(name)` to remove a single table (see pull request #100).
  - Along the way: celebrating 100 issues and pull requests! Thanks everyone for every single contribution!
- Extend API documentation (see issue #96).

4.2.38 v3.1.3 (2016-02-14)

- Fix a bug when using unhashable documents (lists, dicts) with `Query.any` or `Query.all` queries (see a forum post by karibul).

4.2.39 v3.1.2 (2016-01-30)

- Fix a bug when using unhashable documents (lists, dicts) with `Query.any` or `Query.all` queries (see a forum post by karibul).

4.2.40 v3.1.1 (2016-01-23)

- Inserting a dictionary with data that is not JSON serializable doesn't lead to corrupt files anymore (see issue #89).
- Fix a bug in the LRU cache that may lead to an invalid query cache (see issue #87).

4.2.41 v3.1.0 (2015-12-31)

- `db.update(...)` and `db.remove(...)` now return affected document IDs (see issue #83).
- Inserting an invalid document (i.e. not a `dict`) now raises an error instead of corrupting the database (see issue #74).

4.2.42 v3.0.0 (2015-11-13)

- Overhauled Query model:
  - `where('...').contains('...')` has been renamed to `where('...').search('...')`.
  - Support for ORM-like usage: `User = Query(); db.search(User.name == 'John')`.

  - `where('foo')` is an alias for `Query().foo`.
  - `where('foo').has('bar')` is replaced by either `where('foo').bar` or `Query().foo.bar`.
    * In case the key is not a valid Python identifier, array notation can be used: `where('a.b.c')` is now `Query()['a.b.c']`.
  - Checking for the existence of a key has to be done explicitely: `where('foo').exists()`.
- Migrations from v1 to v2 have been removed.
- `SmartCacheTable` has been moved to msiemens/tinydb-smartcache.
- Serialization has been moved to msiemens/tinydb-serialization.
- Empty storages are now expected to return `None` instead of raising `ValueError`. (see issue #67.

4.2.43 v2.4.0 (2015-08-14)

- Allow custom parameters for custom test functions (see issue #63 and pull request #64).

4.2.44 v2.3.2 (2015-05-20)

- Fix a forgotten debug output in the `SerializationMiddleware` (see issue #55).
- Fix an "ignored exception" warning when using the `CachingMiddleware` (see pull request #54)
- Fix a problem with symlinks when checking out TinyDB on OSX Yosemite (see issue #52).

4.2.45 v2.3.1 (2015-04-30)

- Hopefully fix a problem with using TinyDB as a dependency in a `setup.py` script (see issue #51).

4.2.46 v2.3.0 (2015-04-08)

- Added support for custom serialization. That way, you can teach TinyDB to store `datetime` objects in a JSON file :) (see issue #48 and pull request #50)
- Fixed a performance regression when searching became slower with every search (see issue #49)
- Internal code has been cleaned up

4.2.47 v2.2.2 (2015-02-12)

- Fixed a data loss when using `CachingMiddleware` together with `JSONStorage` (see issue #47)

4.2.48 v2.2.1 (2015-01-09)

- Fixed handling of IDs with the JSON backend that converted integers to strings (see issue #45)

4.2.49 v2.2.0 (2014-11-10)

- Extended `any` and `all` queries to take lists as conditions (see pull request #38)
- Fixed an `decode error` when installing TinyDB in a non-UTF-8 environment (see pull request #37)
- Fixed some issues with `CachingMiddleware` in combination with `JSONStorage` (see pull request #39)

4.2.50 v2.1.0 (2014-10-14)

- Added `where(...).contains(regex)` (see issue #32)
- Fixed a bug that corrupted data after reopening a database (see issue #34)

4.2.51 v2.0.1 (2014-09-22)

- Fixed handling of Unicode data in Python 2 (see issue #28).

4.2.52 v2.0.0 (2014-09-05)

Upgrade Notes

Warning: TinyDB changed the way data is stored. You may need to migrate your databases to the new scheme. Check out the *Upgrade Notes* for details.

- The syntax `query in db` has been removed, use `db.contains` instead.
- The `ConcurrencyMiddleware` has been removed due to a insecure implementation (see issue #18). Consider *tinyrecord* instead.
- Better support for working with *Document IDs*.
- Added support for nested comparisons.
- Added `all` and `any` comparisons on lists.
- Added optional :<http://tinydb.readthedocs.io/en/v2.0.0/usage.html#smart-query-cache>‘_.
- The query cache is now a *fixed size LRU cache*.

4.2.53 v1.4.0 (2014-07-22)

- Added `insert_multiple` function (see issue #8).

4.2.54 v1.3.0 (2014-07-02)

- Fixed bug #7: IDs not unique.
- Extended the API: `db.count(where(...))` and `db.contains(where(...))`.
- The syntax `query in db` is now **deprecated** and replaced by `db.contains`.

4.2.55 v1.2.0 (2014-06-19)

- Added `update` method (see issue #6).

4.2.56 v1.1.1 (2014-06-14)

- Merged PR #5: Fix minor documentation typos and style issues.

4.2.57 v1.1.0 (2014-05-06)

- Improved the docs and fixed some typos.
- Refactored some internal code.
- Fixed a bug with multiple `TinyDB?` instances.

4.2.58 v1.0.1 (2014-04-26)

- Fixed a bug in `JSONStorage` that broke the database when removing entries.

4.2.59 v1.0.0 (2013-07-20)

- First official release – consider TinyDB stable now.

4.3 Upgrading to Newer Releases

4.3.1 Version 4.0

- **API changes:**
  - Replace `TinyDB.purge_tables(...)` with `TinyDB.drop_tables(...)`
  - Replace `TinyDB.purge_table(...)` with `TinyDB.drop_table(...)`
  - Replace `Table.purge()` with `Table.trunacte()`
  - Replace `TinyDB(default_table='name')` with `TinyDB.default_table_name = 'name'`
  - Replace `TinyDB(table_class=Class)` with `TinyDB.table_class = Class`
  - If you were using `TinyDB.DEFAULT_TABLE`, `TinyDB.DEFAULT_TABLE_KWARGS`, or `TinyDB.DEFAULT_STORAGE`: Use the new methods for customizing TinyDB described in *How to Extend TinyDB*

4.3.2 Version 3.0

Breaking API Changes

- Querying (see Issue #62):
  - `where('...').contains('...')` has been renamed to `where('...').search('...')`.

  - `where('foo').has('bar')` is replaced by either `where('foo').bar` or `Query().foo.bar`.
    * In case the key is not a valid Python identifier, array notation can be used: `where('a.b.c')` is now `Query()['a.b.c']`.

- Checking for the existence of a key has to be done explicitely: `where('foo').exists()`.
- `SmartCacheTable` has been moved to msiemens/tinydb-smartcache.
- Serialization has been moved to msiemens/tinydb-serialization.
- Empty storages are now expected to return `None` instead of raising `ValueError` (see Issue #67).

4.3.3 Version 2.0

Breaking API Changes

- The syntax `query in db` is not supported any more. Use `db.contains(...)` instead.
- The `ConcurrencyMiddleware` has been removed due to a insecure implementation (see Issue #18). Consider *tinyrecord* instead.

Apart from that the API remains compatible to v1.4 and prior.

For migration from v1 to v2, check out the v2.0 documentation

Python Module Index

t

Index

Symbols

A

C

D

E

F

G