TinyDB Documentation

Release 3.5.0

Markus Siemens

Aug 30, 2017

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

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.

Why Use TinyDB?

- **tiny:** The current source code has 1200 lines of code (with about 40% documentation) and 1000 lines tests. For comparison: Buzhug has about 2500 lines of code (w/o tests), CodernityDB has about 7000 lines of code (w/o 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 2.6 + 2.7 and 3.3 – 3.6 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.

Why Not Use TinyDB?

- **You need advanced features like:**

  - access from multiple processes or threads,
  - creating indexes for tables,
  - a 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.

Getting Started

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:

```
$ python setup.py install
```

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 element's ID. Read more about it here: *Using Element IDs*.

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

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

You can also iter over stored elements:

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

Of course you'll also want to search for specific elements. 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 elements:

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

Recap

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

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

Advanced Usage

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.

Alternative JSON library

As already mentioned, the default storage relies upon Python's JSON module. To improve performance, you can install ujson , an extremely fast JSON implementation. TinyDB will auto-detect and use it if possible.

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')
```

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:
>>> db.search(User.name.matches('[aZ]*'))
>>> 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))
```

When a field contains a list, you also can use the following methods:

```
>>> # Using a query:
>>> # User is member of at least one admin group
>>> db.search(User.groups.any(Group.name == 'admin'))
>>> # User is only member of admin groups
>>> db.search(User.groups.all(Group.name == 'admin'))
```

```
>>> # Using a list of values:
>>> # User is member of at least one group which is 'admin' or 'user'
>>> db.search(User.groups.any(['admin', 'user']))
>>> # User's groups are all either 'admin' or 'user'
>>> db.search(User.groups.all(['admin', 'user']))
```

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.

Recap

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

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

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 an element using `db.insert(...)`. In case you want to insert multiple elements, 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))
```

Updating data

`db.update(fields, query)` only allows you to update an element 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 elements. TinyDB comes with these operations:

- `delete(key)`: delete a key from the element
- `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(element):
...         # do something with the element
...         # ...
...     return transform
...
>>> db.update(your_operation(arguments), query)
```

Retrieving data

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

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

Then of course you can use `db.search(...)` as described in the *Getting Started* section. But sometimes you want to get only one matching element. 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 elements match the query, probably a random one of them will be returned!

Often you don't want to search for elements 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 elements 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 elements
Updating data	
`db.update(operation, ...)`	Update all matching elements with a special operation
Retrieving data	
`len(db)`	Get the number of elements in the database
`db.get(query)`	Get one element matching the query
`db.contains(query)`	Check if the database contains a matching element
`db.count(query)`	Get the number of matching elements

Using Element IDs

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

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

In addition you can get the ID of already inserted elements using `element.eid`. This works both with `get` and `all`:

```
>>> el = db.get(User.name == 'John')
>>> el.eid
3
>>> el = db.all()[0]
>>> el.eid
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}, eids=[1, 2])
>>> db.contains(eids=[1])
True
>>> db.remove(eids=[1, 2])
>>> db.get(eid=3)
{...}
```

Recap

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

Getting an element's ID	
`db.insert(...)`	Returns the inserted element's ID
`db.insert_multiple(...)`	Returns the inserted elements' ID
`element.eid`	Get the ID of an element fetched from the db
Working with IDs	
`db.get(eid=...)`	Get the element with the given ID
`db.contains(eids=[...])`	Check if the db contains elements with one of the given IDs
`db.update({...}, eids=[...])`	Update all elements with the given IDs
`db.remove(eids=[...])`	Remove all elements with the given IDs

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.purge_table('table_name')
```

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

```
>>> db.purge_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 either passing `default_table` to the `TinyDB` constructor or by setting the `DEFAULT_TABLE` class variable to modify the default table name for all instances:

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

Query Caching

TinyDB caches query result for performance. 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.

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.

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

```
>>> TinyDB.DEFAULT_STORAGE = MemoryStorage
```

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()
```

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

How to Extend TinyDB

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

1. custom storage,
2. custom middleware, and
3. custom table classes.

Let's look at them in this order.

Write Custom Storage

First, we have support for custom storage. By default TinyDB comes with an in-memory storage mechanism and a JSON file storage mechanism. 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)

    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)
# ...
```

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 any 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=TinyDB.DEFAULT_STORAGE):
```

```
        # Any middleware *has* to call the super constructor
        # with storage_cls
        super(CustomMiddleware, self).__init__(storage_cls)

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

        for table_name in data:
            table = data[table_name]

            for element_id in table:
                item = table[element_id]

                if item == {}:
                    del table[element_id]

        return data

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

            for element_id in table:
                item = table[element_id]

                if item == {}:
                    del table[element_id]

        self.storage.write(data)

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

Two remarks:

1. You have to use the `super(...)` call as shown in the example. To run your own initialization, add it below the `super(...)` call.
2. This is an example for middleware, not an example for clean code. Don't use it as shown here without at least refactoring the loops into a separate method.

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`).

Creating a Custom Table Classes

Custom storage and middleware are useful if you want to modify the way TinyDB stores its data. But there are cases where you want to modify how TinyDB itself behaves. For that use case TinyDB supports custom table classes. Internally TinyDB creates a `Table` instance for every table that is used. You can overwrite which class is used by setting `TinyDB.table_class` before creating a `TinyDB` instance. This class has to support the *Table API*. The best way to accomplish that is to subclass it:

```
from tinydb.database import Table

class YourTableClass(Table):
    pass  # Modify original methods as needed
```

For an more advanced example, see the source of the tinydb-smartcache extension.

Extensions

Here are some extensions that might be useful to you:

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.

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.

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.

tinydb-serialization

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

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 elements so the cache doesn't get invalidated. It's useful if you perform lots of queries while the data changes only little.

CHAPTER 3

API Reference

API Documentation

tinydb.database

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

The main class of TinyDB.

Gives access to the database, provides methods to insert/search/remove and getting tables.

DEFAULT_STORAGE

alias of JSONStorage

__getattr__(*name*)

Forward all unknown attribute calls to the underlying standard table.

__init__(**args*, ***kwargs*)

Create a new instance of TinyDB.

All arguments and keyword arguments will be passed to the underlying storage class (default: *JSONStorage*).

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

__iter__()

Iter over all elements from default table.

__len__()

Get the total number of elements in the default table.

```
>>> db = TinyDB('db.json')
>>> len(db)
0
```

close()
Close the database.

purge_table(*name*)
Purge a specific table from the database. **CANNOT BE REVERSED!**

Parameters **name** (*str*) – The name of the table.

purge_tables()
Purge all tables from the database. **CANNOT BE REVERSED!**

table(*name='_default'*, ***options*)
Get access to a specific table.

Creates a new table, if it hasn't been created before, otherwise it returns the cached `Table` object.

Parameters

- **name** (*str*) – The name of the table.
- **cache_size** – How many query results to cache.

table_class
alias of `Table`

tables()
Get the names of all tables in the database.

Returns a set of table names

Return type set[str]

class tinydb.database.**Table**(*storage*, *name*, *cache_size=10*)
Represents a single TinyDB Table.

__init__(*storage*, *name*, *cache_size=10*)
Get access to a table.

Parameters

- **storage** (*StorageProxy*) – Access to the storage
- **name** – The table name
- **cache_size** – Maximum size of query cache.

__iter__()
Iter over all elements stored in the table.

Returns an iterator over all elements.

Return type listiterator[*Element*]

__len__()
Get the total number of elements in the table.

all()
Get all elements stored in the table.

Returns a list with all elements.

Return type list[*Element*]

clear_cache()
Clear the query cache.

A simple helper that clears the internal query cache.

contains(*cond=None*, *eids=None*)
Check wether the database contains an element matching a condition or an ID.

If `eids` is set, it checks if the db contains an element with one of the specified.

Parameters

- **cond** (`Query`) – the condition use
- **eids** – the element IDs to look for

count(*cond*)
Count the elements matching a condition.

Parameters **cond** (`Query`) – the condition use

get(*cond=None*, *eid=None*)
Get exactly one element specified by a query or and ID.

Returns `None` if the element doesn't exist

Parameters

- **cond** (`Query`) – the condition to check against
- **eid** – the element's ID

Returns the element or None

Return type Element | None

insert(*element*)
Insert a new element into the table.

Parameters **element** – the element to insert

Returns the inserted element's ID

insert_multiple(*elements*)
Insert multiple elements into the table.

Parameters **elements** – a list of elements to insert

Returns a list containing the inserted elements' IDs

name
Get the table name.

process_elements(*func*, *cond=None*, *eids=None*)
Helper function for processing all elements specified by condition or IDs.

A repeating pattern in TinyDB is to run some code on all elements that match a condition or are specified by their ID. This is implemented in this function. The function passed as `func` has to be a callable. It's first argument will be the data currently in the database. It's second argument is the element ID of the currently processed element.

See: *update()*, *remove()*

Parameters

- **func** – the function to execute on every included element. first argument: all data second argument: the current eid
- **cond** – elements to use, or
- **eids** – elements to use

Returns the element IDs that were affected during processed

purge()
Purge the table by removing all elements.

remove(*cond=None*, *eids=None*)
Remove all matching elements.

Parameters

- **cond** (*query*) – the condition to check against
- **eids** (*list*) – a list of element IDs

Returns a list containing the removed element's ID

search(*cond*)
Search for all elements matching a 'where' cond.

Parameters **cond** (Query) – the condition to check against

Returns list of matching elements

Return type list[*Element*]

update(*fields*, *cond=None*, *eids=None*)
Update all matching elements to have a given set of fields.

Parameters

- **fields** (*dict | dict -> None*) – the fields that the matching elements will have or a method that will update the elements
- **cond** (*query*) – which elements to update
- **eids** (*list*) – a list of element IDs

Returns a list containing the updated element's ID

class tinydb.database.**Element**(*value=None*, *eid=None*, ***kwargs*)
Represents an element stored in the database.

This is a transparent proxy for database elements. It exists to provide a way to access an element's id via `el.eid`.

eid
The element's id

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:

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

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

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

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

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

__ge__(*rhs*)
Test a dict value for being greater than or equal to another value.

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

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

__gt__(*rhs*)
Test a dict value for being greater than another value.

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

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

__le__(*rhs*)
Test a dict value for being lower than or equal to another value.

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

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

__lt__(*rhs*)
Test a dict value for being lower than another value.

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

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

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

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

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

all(*cond*)
Checks if a condition is met by any element 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])
# Match f1 that contains any element from [1, 2, 3]
```

Matches:

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

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

any(*cond*)

Checks if a condition is met by any element 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])
# Match f1 that contains any element from [1, 2, 3]
```

Matches:

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

param cond Either a query that at least one element has to match or a list of which at least one element has to be contained in the tested element.

•

exists()

Test for a dict where a provided key exists.

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

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

matches(*regex*)

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

search(*regex*)
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

test(*func*, **args*)
Run a user-defined test function against a dict value.

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

Parameters

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

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*, *create_dirs=False*, ***kwargs*)
Store the data in a JSON file.

__init__(*path*, *create_dirs=False*, ***kwargs*)
Create a new instance.

Also creates the storage file, if it doesn't exist.

Parameters **path** (*str*) – Where to store the JSON data.

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

__init__()
Create a new instance.

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=<class 'tinydb.storages.JSONStorage'>*)
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.

flush()
Flush all unwritten data to disk.

CHAPTER 4

Additional Notes

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 :)

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.

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 ...
    """
```

```
    # implementation ...
```

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.

Changelog

Version Numbering

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

v3.5.0 (2017-08-30)

- Expose the table name via `table.name` (see issue #148).
- Allow better subclassing of the `TinyDB` class (see pull request #150).

v3.4.1 (2017-08-23)

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

v3.4.0 (2017-08-08)

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

v3.3.1 (2017-06-27)

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

v3.3.0 (2017-06-05)

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

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).

v3.2.2 (2017-01-16)

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

v3.2.1 (2016-06-29)

- Fix a bug with queries on elements 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).

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).

v3.1.3 (2016-02-14)

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

v3.1.2 (2016-01-30)

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

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).

v3.1.0 (2015-12-31)

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

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.

v2.4.0 (2015-08-14)

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

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).

v2.3.1 (2015-04-30)

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

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

v2.2.2 (2015-02-12)

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

v2.2.1 (2015-01-09)

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

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)

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)

v2.0.1 (2014-09-22)

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

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 *Element 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*.

v1.4.0 (2014-07-22)

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

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`.

v1.2.0 (2014-06-19)

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

v1.1.1 (2014-06-14)

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

v1.1.0 (2014-05-06)

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

v1.0.1 (2014-04-26)

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

v1.0.0 (2013-07-20)

- First official release – consider TinyDB stable now.

Upgrading to Newer Releases

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).

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

T

U

W