TinyDB Documentation

Release 2.3.2

Markus Siemens

May 20, 2015

Contents

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

```
>>> from tinydb import TinyDB, where
>>> db = TinyDB('path/to/db.json')
>>> db.insert({'int': 1, 'char': 'a'})
>>> db.search(where('int') == 1)
[{'int': 1, 'char': 'a'}]
```

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 using TinyDB?

- **tiny:** The current source code has 1200 (with about 40% documentation) lines of code (+ 600 lines tests). For comparison: Buzhug has about 2000 lines of code (w/o tests), CodernityDB has about 8000 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 – 3.4 and PyPy:** TinyDB works on all modern versions of Python and PyPy.
- **easily extensible:** You can easily extend TinyDB by writing new storages or modify the behaviour of storages with Middlewares.
- **nearly 100% test coverage:** If you don't count that `__repr__` methods and some abstract methods are not tested, TinyDB has a test coverage of 100%.

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 using TinyDB?

- You need **advanced features** like multiple indexes, an HTTP server, relationships, or similar.
- You are really concerned about **high 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 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:

```
$ python setup.py 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, where
>>> 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})
1
>>> db.insert({'type': 'peach', 'count': 3})
2
```

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'}]
```

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

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

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

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

In the same manner you can also remove elements:

```
>>> db.remove(where('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
`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	
`where('field') == 2`	Match any element that has a key `field` with value `== 2` (also possible: `!= > >= < <=`)

1.3 Advanced Usage

1.3.1 Remarks on Storages

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 libary

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.

1.3.2 Handling Data

With that out of the way, let's start with inserting, updating and retrieving 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([{'int': 1, 'char': 'a'}, {'int': 1, 'char': 'b'}])
>>> 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 it's values. But sometimes you may need to e.g. remove one field or increment it's value. In that case you can pass a function instead of `fields`:

```
>>> from tinydb.operations import delete
>>> db.update(delete('key1'), where('key') == 'value')
```

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)`: incremenet the value of a key
- `decrement(key)`: decremenet the value of a key

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(where('value') == 1)[0]
... except IndexError:
...     pass
```

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

```
>>> db.get(where('value') == 1)
{'int': 1, 'value': 1}
>>> db.get(where('value') == 100)
None
```

> **Caution:** If multiple elements match the query, propably 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(where('char') == 'a')
```

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

```
>>> db.count(where('int') == 1)
3
```

Recap

Let's summarize the ways to handle data:

Inserting data	
`db.insert_multiple(...)`	Insert multiple elements
Updatingg 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

1.3.3 Using Element IDs

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

```
>>> db.insert({'value': 1})
3
>>> db.insert_multiple([{...}, {...}, {...}])
[4, 5, 6]
```

In addition you can get the ID of already inserted elements using `element.eid`:

```
>>> el = db.get(where('value') == 1)
>>> el.eid
3
```

Different TinyDB methods also work with IDs, namely: `update`, `remove`, `contains` and `get`.

```
>>> 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 elemtent 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

1.3.4 Queries

TinyDB lets you use a rich set of queries. In the Getting Started you've learned about the basic comparisons (==, <, >, ...). In addition to that TinyDB enables you run logical operations on queries.

```
>>> # Negate a query:
>>> db.search(~ where('int') == 1)
```

```
>>> # Logical AND:
>>> db.search((where('int') == 1) & (where('char') == 'b'))
[{'int': 1, 'char': 'b'}]
```

```
>>> # Logical OR:
>>> db.search((where('char') == 'a') | (where('char') == 'b'))
[{'int': 1, 'char': 'a'}, {'int': 1, 'char': 'b'}]
```

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

You also can search for elements where a specific key exists:

```
>>> db.search(where('char'))
[{'int': 1, 'char': 'a'}, {'int': 1, 'char': 'b'}]
```

Advanced queries

In addition to these checks TinyDB supports checking against a regex or a custom test function:

```
>>> # Regex:
>>> db.search(where('char').matches('[aZ]*'))
[{'int': 1, 'char': 'abc'}, {'int': 1, 'char': 'def'}]
>>> db.search(where('char').contains('b+'))
[{'int': 1, 'char': 'abbc'}, {'int': 1, 'char': 'bb'}]
```

```
>>> # Custom test:
>>> test_func = lambda c: c == 'a'
>>> db.search(where('char').test(test_func))
[{'char': 'a', 'int': 1}]
```

Nested Queries

You can insert nested elements into your database:

```
>>> db.insert({'field': {'name': {'first_name': 'John', 'last_name': 'Doe'}}})
```

To search for a nested field, use `where('field').has(...)`. You can apply any queries you already know to this selector:

```
>>> db.search(where('field').has('name'))
[{'field': ...}]
>>> db.search(where('field').has('name').has('last_name') == 'Doe')
[{'field': ...}]
```

You also can use lists inside of elements:

```
>>> db.insert({'field': [{'val': 1}, {'val': 2}, {'val': 3}]})
```

Using `where('field').any(...)` and `where('field').all(...)` you can specify checks for the list's items using either a nested query or a sequence such as a list. They behave similarly to Python's *any* and *all*:

```
>>> # Nested Query:
>>> db.search(where('field').any(where('val') == 1))
True
>>> db.search(where('field').all(where('val') > 0))
True
```

```
>>> # List:
>>> db.search(where('field').any([{'val': 1}, {'val': 4}]))
True
>>> db.search(where('field').all([{'val': 1}, {'val': 4}]))
False
>>> db.search(where('field').all([{'val': 1}, {'val': 3}]))
True
```

Recap

Again, let's recapitulate the query operations:

Queries	
`where('field').matches(regex)`	Match any element matching the regular expression
`where('field').contains(regex)`	Match any element with a matching substring
`where('field').test(func)`	Matches any element for which the function returns `True`
Combining Queries	
`~ query`	Match elements that don't match the query
`(query1) & (query2)`	Match elements that match both queries
`(query1) | (query2)`	Match elements that match one of the queries
Nested Queries	
`where('field').has('field')`	Match any element that has the specified item. Perform more queries on this selector as needed
`where('field').any(query)`	Match any element where 'field' is a list where one of the items matches the subquery
`where('field').all(query)`	Match any element where 'field' is a list where all items match the subquery

1.3.5 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}]
```

To remove all tables from a database, use:

```
>>> db.purge_tables()
```

Note: TinyDB uses a table named `_default` as default table. All operations on the database object (like `db.insert(...)`) operate on this table.

You can get a list with the names of all tables in your database:

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

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.

Smart Query Cache

If you perform lots of queries while the data changes only little, you may enable a smarter query cache. It updates the query cache when inserting/removing/updating elements so the cache doesn't get invalidated.

```
>>> table = db.table('table_name', smart_cache=True)
```

1.3.6 Storages & Middlewares

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)
```

Middlewares

Middlewares wrap around existing storages 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 middlewares:

```
>>> 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()
```

1.3.7 What's next

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

- Want to learn how to customize TinyDB (custom serializers, 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

2.1.1 Write a Serializer

TinyDB's default JSON storage is fairly limited when it comes to supported data types. If you need more flexibility, you can implement a Serializer. This allows TinyDB to handle classes it couldn't serialize otherwise. Let's see how a Serializer for `datetime` objects could look like:

```
from datetime import datetime

class DateTimeSerializer(Serializer):
    OBJ_CLASS = datetime  # The class this serializer handles

    def encode(self, obj):
        return obj.strftime('%Y-%m-%dT%H:%M:%S')

    def decode(self, s):
        return datetime.strptime(s, '%Y-%m-%dT%H:%M:%S')
```

To use the new serializer, we need to use the serialization middleware:

```
>>> from tinydb.storages import JSONStorage
>>> from tinydb.middlewares import SerializationMiddleware
>>>
>>> serialization = SerializationMiddleware()
>>> serialization.register_serializer(DateTimeSerializer(), 'TinyDate')
>>>
>>> db = TinyDB('db.json', storage=serialization)
>>> db.insert({'date': datetime(2000, 1, 1, 12, 0, 0)})
>>> db.all()
[{'date': datetime.datetime(2000, 1, 1, 12, 0)}]
```

2.1.2 Write a custom Storage

By default TinyDB comes with a 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:
            data = yaml.safe_load(handle.read())  # (2)

        if data is None:  # (3)
            raise ValueError

    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 throw a `ValueError` 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)
# ...
```

2.1.3 Write a custom Middleware

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

Because middlewares act 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 a 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 a storate with this new middleware, we use it like this:

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

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

2.2 Extensions

2.2.1 `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.

2.2.2 `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.

CHAPTER 3

API Reference

3.1 API Documentation

3.1.1 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

__enter__()

See *Table.__enter__()*

__exit__(**args*)

See *Table.__exit__()*

__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.

__len__()

Get the total number of elements in the DB.

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

purge_tables()

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

table(*name='_default'*, *smart_cache=False*, ***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.
- **smart_cache** – Use a smarter query caching. See *tinydb.database.SmartCacheTable*
- **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**(*name*, *db*, *cache_size=10*)
Represents a single TinyDB Table.

__enter__()
Allow the database to be used as a context manager.

Returns the table instance

__exit__(**args*)
Try to close the storage after being used as a context manager.

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

Parameters

- **name** (*str*) – The name of the table.
- **db** (tinydb.database.TinyDB) – The parent database.
- **cache_size** – Maximum size of query cache.

__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]

close(**args*)
Try to close the storage after being used as a context manager.

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

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
- **eids** – elements to use

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

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, int) -> 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

class tinydb.database.**SmartCacheTable** (*name*, *db*, *cache_size=10*)
A Table with a smarter query cache.

Provides the same methods as *Table*.

The query cache gets updated on insert/update/remove. Useful when in cases where many searches are done but data isn't changed often.

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

3.1.2 tinydb.queries

class tinydb.queries.**Query** (*key*)
Provides methods to do tests on dict fields.

Any type of comparison will be called in this class. In addition, it is aliased to `where` to provide a more intuitive syntax.

When not using any comparison operation, this simply tests for existence of the given key.

__and__ (*other*)
Combines this query and another with logical and.

Example:

```
>>> (where('f1') == 5) & (where('f2') != 2)
('f1' == 5) and ('f2' != 2)
```

Return type QueryAnd

__call__ (*element*)
Run the test on the element.

Parameters **element** (*dict*) – The dict that we will run our tests against.

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

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

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

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

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

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

__invert__()
Negates a query.

```
>>> ~(where('f1') >= 42)
not ('f1' >= 42)
```

Return type tinydb.queries.QueryNot

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

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

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

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

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

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

__or__(*other*)
Combines this query and another with logical or.

Example:

```
>>> (where('f1') == 5) | (where('f2') != 2)
('f1' == 5) or ('f2' != 2)
```

Return type QueryOr

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).

```
>>> where('f1').all(where('f2') == 1)
'f1' all have 'f2' == 1
```

Matches:

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

```
>>> where('f1').all([{'f2': 1}, {'f3': 2}])
'f1' all have [{'f2': 1}, {'f3': 2}]
```

Matches:

```
{'f1': [{'f2': 1}, {'f3': 2}]}
{'f1': [{'f2': 1}, {'f3': 2}, {'f4': 3}]}
```

Parameters **cond** – The condition to check

Return type *tinydb.queries.Query*

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).

```
>>> where('f1').any(where('f2') == 1)
'f1' has any 'f2' == 1
```

Matches:

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

```
>>> where('f1').any([1, 2, 3])
'f1' has any [1, 2, 3]
```

Matches:

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

Parameters **cond** – The condition to check

Return type *tinydb.queries.Query*

contains(*regex*)

Run a regex test against a dict value (only substring has to match).

```
>>> where('f1').contains(r'\d+')
'f1' ~= \d+
```

Parameters **regex** – The regular expression to pass to `re.search`

Return type QueryRegex

has(*key*)

Run test on a nested dict.

```
>>> where('x').has('y') == 2
has 'x' => ('y' == 2)
```

Matches:

```
{'x': {'y': 2}}
```

Parameters **key** – the key to search for in the nested dict

Return type QueryHas

matches(*regex*)

Run a regex test against a dict value (whole string has to match).

```
>>> where('f1').matches(r'^\w+$')
'f1' ~= ^\w+$
```

Parameters **regex** – The regular expression to pass to `re.match`

Return type QueryRegex

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

```
>>> def test_func(val):
...     return val == 42
...
>>> where('f1').test(test_func)
'f1'.test(<function test_func at 0xXXXXXXXX>)
```

Parameters **func** – The function to run. Has to accept one parameter and return a boolean.

Return type QueryCustom

3.1.3 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*)
Store the data in a JSON file.

__init__(*path*)
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.

3.1.4 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

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 ...
    """

    # 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 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).

4.2.3 v2.3.1 (2015-04-30)

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

4.2.4 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

4.2.5 v2.2.2 (2015-02-12)

- Fixed a data loss when using `CachingMiddleware` together with `JSONStorage` (see issue #45)

4.2.6 v2.2.1 (2015-01-09)

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

4.2.7 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)

4.2.8 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)

4.2.9 v2.0.1 (2014-09-22)

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

4.2.10 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 *smart query caching*.
- The query cache is now a *fixed size lru cache*.

4.2.11 v1.4.0 (2014-07-22)

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

4.2.12 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`.

4.2.13 v1.2.0 (2014-06-19)

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

4.2.14 v1.1.1 (2014-06-14)

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

4.2.15 v1.1.0 (2014-05-06)

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

4.2.16 v1.0.1 (2014-04-26)

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

4.2.17 v1.0.0 (2013-07-20)

- First official release – consider TinyDB stable now.

4.3 Upgrading to Newer Releases

4.3.1 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.

Migration

To improve the handling of IDs TinyDB changed the way it stores data (see Issue #13 for details). Opening an database from v1.4 or prior will most likely result in an exception:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "tinydb\database.py", line 49, in __init__
    self._table = self.table('_default')
  File "tinydb\database.py", line 69, in table
    table = table_class(name, self, **options)
  File "tinydb\database.py", line 171, in __init__
    self._last_id = int(sorted(self._read().keys())[-1])
  File "tinydb\database.py", line 212, in _read
    data[eid] = Element(data[eid], eid)
TypeError: list indices must be integers, not dict
```

In this case you need to migrate the database to the recent scheme. TinyDB provides a migration script for the default JSON storage:

```
$ python -m tinydb.migrate db1.json db2.json
Processing db1.json ... Done
Processing db2.json ... Done
```

Migration with a custom storage

If you have database files that have been written using a custom storage class, you can write your own migration script that calls `tinydb.migrate.migrate`:

tinydb.migrate.**migrate**(**args*, ***kwargs*)

Migrate a database to the scheme used in v2.0.

To migrate a db that uses a custom storage, use

```
>>> from tinydb.migrate import migrate
>>> args = [...]  # args for your storage class
>>> kwargs = {...}  # kwargs for your storage class
>>> migrate(*args, **kwargs, storage=YourStorageClass)
True
```

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

Returns `True` if the db has been migrated, `False` if the db didn't need a migration.

t

Symbols

A

C

D

E

F

G

H

I

J

M

P

Q

R

S

T

U

W