TinyDB Documentation

Markus Siemens

Jun 09, 2020

Contents

DeepClean is a multimodal data pre-processing and cleaning package written in python.

CHAPTER 1

Features

- Single package for multi modal data pre-processing and cleaning.
- Fast Experimentation
- Modularity in the data (processing) pipeline
- **Implements techniques like:**
  - Normalization
  - Tokenization
  - Augmentation
  - Denoising
  - and much more.

```
>>> from deepclean.image import Augmentation
>>> aug = Augmentation('path/to/image')
```

```
>>> from deepclean.text import Tokenization
>>> tk = Tokenization('path/to/text')
```

CHAPTER 2

User's Guide

2.1 Introduction

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

2.1.1 Why Use DeepClean?

- **multimodal data:** If you have multimodal data - visual, text, speech.
- **fast experimentation:** If you need to quickly do some analysis or do proof of concept project.
- **optimized for your happiness:** DeepClean is designed to be simple and fun to use by providing a simple and clean API.
- **written in pure Python:** Truly pythonic.
- **100% test coverage:** No explanation needed.

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

2.1.2 Why Not Use DeepClean?

- **You need advanced features like:**
  - Segmentation
- You are really concerned about **performance** and need a high speed computation.

To put it plainly: If you need advanced features or high performance, DeepClean is the wrong package for you.

2.2 Getting Started

2.2.1 Installing DeepClean

To install TinyDB from PyPI, run:

```
$ pip install deepclean
```

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

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

2.2.2 Basic Usage

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

```
>>> from deepclean import Normalizer, Tokenization
>>> nm = Normalizer(text)
```

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

Recap

2.3 Advanced Usage

2.3.1 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.

2.3.2 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))
```

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']}]
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # 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({
        '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

2.3.3 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))
```

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)
```

2.3.4 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.

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
```

Replacing data

Another occasionally useful operation is to replace a list of documents. If you have a list of documents with IDs (see *document_ids*), you can pass them to `db.write_back(list)`:

```
>>> docs = db.search(User.name == 'John')
[{name: 'John', age: 12}, {name: 'John', age: 44}]
>>> for doc in docs:
...     doc['name'] = 'Jane'
>>> db.write_back(docs)  # Will update the documents we retrieved
>>> docs = db.search(User.name == 'John')
[]
```

(continues on next page)

(continued from previous page)

```
>>> docs = db.search(User.name == 'Jane')
[{name: 'Jane', age: 12}, {name: 'Jane', age: 44}]
```

Alternatively you can pass a list of documents along with a list of document IDs to achieve the same goal. In this case, the length of the document list and the ID list has to be equal.

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
`db.write_back(docs)`	Replace all documents with the updated versions
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

2.3.5 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
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_ids=[1])
True
>>> db.remove(doc_ids=[1, 2])
```

(continues on next page)

(continued from previous page)

```
>>> 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:

2.3.6 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.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. Also, you can set `cache_size` to 0 to disable it.

2.3.7 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 variable:

```
>>> TinyDB.DEFAULT_STORAGE = 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()
```

2.3.8 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 extend and extensions.
- Want to study the API in detail? Read *API Documentation*.
- Interested in contributing to the TinyDB development guide? Go on to the contribute.

CHAPTER 3

API Reference

3.1 API Documentation

3.1.1 `tinydb.database`

3.1.2 `tinydb.queries`

3.1.3 `tinydb.storage`

3.1.4 `tinydb.middlewares`